**UNIVERSITÉ DE MONTRÉAL**

# REUSE AND AUTOMATIC GENERATION OF TESTBENCHES FOR EFFECTIVE HARDWARE VERIFICATION

JIAHONG WANG

DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

AVRIL 2003

**UNIVERSITÉ DE MONTRÉAL**

**ÉCOLE POLYTECHNIQUE DE MONTRÉAL**

CE MÉMOIRE INTITULÉ:

**REUSE AND AUTOMATIC GENERATION OF TESTBENCHES FOR EFFECTIVE HARDWARE VERIFICATION**

Présenté par : WANG Jiahong

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen composé de:

M. SAVARIA Yvon, Ph.D., président

M. BOIS Guy, Ph.D., directeur

M. ABOULHAMID El Mostapha, Ph.D., membre

## ACKNOWLEDGMENTS

Lots of people have helped during my stay at GRM. I would like to thank them for their support during my research:

- Firstly and foremost, I would like to thank my director, Prof. Guy Bois, for entrusting me with this project and for being there whenever needed. His patient guidance and encouragement over the past two years has made this a very enjoyable experience.
- I would like to express my deep gratitude to my co-director, Prof. Yvon Savaria, for his many ideas, help and support. Without whose guidance, the following work would not have been possible.
- Thanks a lot to Pascal Nsame, he was the one who first introduced me to the field verification and the base theory in my thesis, *Functional Abstraction,* is came from his idea. I am very glad to continue develop this method.
- Thanks to Sébastien Regimbal, Jean-François Lemire, in discussing with me the different aspects of this project and their friendship.
- It would not have been as much fun without all my friends in the office.
- And last but certainly not least, I would like to express my enormous gratitude towards my family. My parents, Wang Qi and Ma Zhenrong, for their love and support. And Jing, thank you for all you have done for me.

# RÉSUMÉ

Il est bien connu que la vérification fonctionnelle est devenue le goulot d'étranglement principal du processus de conception à cause de la complexité élevée des circuits modernes. Il y a un besoin renforcé de trouver une solution pratique pour réduire le temps requis pour la vérification fonctionnelle. Une nouvelle méthodologie et des outils de vérification sont développés et utilisés au cours de la conception de ce projet. La réutilisation est un concept très populaire dans le développement de dispositifs complexes et une méthodologie de réutilisation de vérification est un point clé pour réduire le temps requis pour l'étape de la vérification. Un outil de génération automatisée peut contribuer à réduire le temps pour permettre aux ingénieurs d'être libérés des tâches de génération et de correction de code.

Ce mémoire apporte un survol général de la vérification fonctionnelle, y compris une analyse des méthodologies existantes, des outils développés et de la littérature sur la vérification fonctionnelle. Nous nous concentrons particulièrement sur la réutilisation de *testbenchs*. Nous présentons une méthode d'abstraction de fonctions (FA), pour produire des *testbenchs* réutilisables au niveau fonctionnel. Par la suite, une méthode de capture des spécifications fonctionnelles de systèmes avec le langage SDL est présentée. L'abstraction des systèmes est commune à tous les modules conçus dans la même spécification.

Cette démarche permet d'établir des *testbenchs* fonctionnels conçus pour la réutilisation. En adaptant les techniques FA et le langage SDL, nous proposons une méthodologie qui permet d'établir des *testbenchs* pour des composants fonctionnels réutilisables. En outre, la technique est basée sur des règles et employée pour développer un outil pour mettre en application cette méthodologie. Les règles définissent les informations qui sont nécessaires pour établir et traiter les *testbenchs*.

Le résultat de cette recherche est la création d'un outil de génération automatique de *testbenchs* qui rencontre les objectifs de notre méthodologie pour établir des composants de *testbench* réutilisables avec un langage de vérification de matériel, le langage *e*.

## ABSTRACT

It is well known that functional verification has become the major bottleneck of the entire design process due to the high complexity of modern circuit designs. There is an emerging need for a practical solution to reduce the functional verification time. New verification methodology and tools are being developed and are well used in the process of project design. As design reuse has been very popular in complex device development, a methodology of verification reuse is a key point to reduce time in verification stage. An automated generation tool can contribute to reduce verification time by relieving engineers from code generation and debugging.

This thesis provides a general review of functional verification, which analyzes existing methodologies. We focus on testbench reuse. We present a method, *Function Abstraction* (FA), to generate reusable testbenches at functional level. Then, a method is presented to capture system functionality with the system description language (SDL). System functional abstraction is common to all modules designed from the same specification. That makes it possible to build testbenches at this functional level aimed to reuse testbenches. Adapting the techniques FA and SDL language, we introduce a methodology to build functional reusable testbench components. In addition, a rule-based technique is used to develop a tool to implement this methodology. Rules define which information is needed to build testbenches and how to process this information.

The result of this research is a testbench generation tool that follows our reusable testbench generation methodology, to build reusable testbench components with the *e* language, a *Hardware Verification Language* (HVL).

# CONDENSE EN FRANÇAIS

## RÉUTILISATION ET GÉNÉRATION AUTOMATIQUE DE BANCS D'ESSAI POUR LA VÉRIFICATION MATÉRIELLE EFFICACE

Ce mémoire présente une approche pour automatiser la génération de bancs d'essai réutilisables. Dans la conception des circuits numériques, la phase la plus longue est certainement celle de la vérification fonctionnelle des spécifications. C'est pourquoi nous proposons une méthodologie pour créer des bancs d'essai réutilisables. Nous accompagnons cette méthodologie d'un outil que nous avons implanté et qui facilite ce processus. La méthodologie proposée se base sur l'abstraction fonctionnelle des systèmes. L'information récupérée de cette abstraction envoyée à l'entrée de l'outil, qui produit alors comme résultats des composants de bancs d'essai réutilisables. Les spécifications sont capturées à l'aide d'un langage spécialisé et combiné à une technique de règles (*rules*) qu'on lie à l'outil. C'est ainsi que nous présentons une méthodologie pour la génération de bancs d'essai fonctionnels en relation avec une méthode pour la capture des abstractions de haut niveau d'un design.

### 1. Introduction

Il est généralement reconnu que la vérification fonctionnelle est l'étape la plus gourmande dans le cycle de conception des systèmes. Selon un récent sondage effectué par notre groupe de recherche [2], le développement de bancs d'essai complexes requiert plus de 50% du temps total d'un flot de design et ce pour plus de 85% des répondants. De

même, seulement 23% des répondants utilisent une méthodologie définie pour créer des bancs d'essai Enfin, seulement 35% adoptent une politique de réutilisation des composants de bancs d'essai

Les langages et les outils disponibles jouent un rôle important pour la conception des systèmes, mais ce rôle est beaucoup moins important pour le processus de vérification. Pourtant, la véritable clef pour une approche plus efficace de la vérification fonctionnelle des systèmes nécessitent l'approbation d'une nouvelle méthodologie [3]. Il apparaît alors intéressant de développer une nouvelle méthodologie qui vise à utiliser un outil pour assister la génération de bancs d'essais. Tel sera notre principal but pour ce projet.

Noter groupe de recherche a déjà présenté une méthodologie de partitionnement par aspects [4], qui promeut la réutilisation en vérification matérielle. Cette méthodologie permet de construire efficacement un environnement de vérification et de tests basés sur une technique de design orientée objets. Les modules de bancs d'essai créés par cette méthode sont réutilisables et peuvent être raffinés. Les bancs d'essai créés pour un IP en particulier, peuvent être réutilisés lorsque la vérification change de niveau d'abstraction (du niveau bloc au niveau système par exemple). Par contre, les bancs d'essai peuvent difficilement être réutilisés par d'autres implantations de IP. Nous recherchons des méthodes qui permettraient de réutiliser les bancs d'essai qui ne sont pas seulement basés sur un IP mais sur une multitude de IP.

Pour pouvoir réutiliser un banc d'essai d'un IP à un autre, le modèle utilisé doit être le plus générique possible. En d'autres termes, il faut capturer le caractère commun des DUT (*design under test*) utilisés. Le modèle de banc d'essai devra donc utiliser tous les caractères communs rencontrés. Cette information qui est requise pour la création des bancs d'essai est constituée de la fonctionnalité et de la structure d'un design. Évidemment, il ne sera pas requis de retirer cette information d'un design de niveau RTL (*Register Transfer Level*) parce que ces implantations varient d'un projet à l'autre. Les mêmes fonctions auraient différentes implantations à travers différents projets, ce qui rend difficile de récupérer cette information commune et générique pour une fonction donnée. Une façon efficace de capturer la fonctionnalité peut être obtenue de la description d'une spécification de haut niveau. Puisque toutes les implantations se réfèrent à ce niveau d'abstraction, ces fonctionnalités abstraites sont donc les mêmes d'un projet à l'autre. Conséquemment, les modèles de bancs d'essai qui utilisent une fonctionnalité abstraite sont réutilisables. Cependant, une contrainte liée à cette approche est que les modèles de bancs d'essai sont seulement réutilisables dans une même famille de design et projets (basées sur les mêmes spécifications). Pour résoudre ce problème, il faut lever le niveau d'abstraction pour englober le plus d'implantations différentes possibles.

Nous avons développé l'outil *Reusable Testbench Generation Tool set* (RTGT) qui est basé sur notre méthodologie de réutilisation de bancs d'essai. Cet outil permet la génération automatique de ces derniers, ce qui accélère grandement leur création en

général. Dans notre méthodologie, SDL (*Specification and Description Language*) est utilisé pour capturer la fonctionnalité des systèmes. Ce sont des spécifications textuelles de SDL qui devront être données en entrée à l'outil RTGT.

## 2. Type de réutilisation de bancs d'essai

Après plusieurs expérimentation en réutilisation de composants au niveau de la vérification, nous croyons que cette classification est juste, (mais sans doute incomplète). Les bancs d'essai peuvent être divisés en deux groupes selon leurs critères de réutilisation.

Type 1 : Composants réutilisables pour une implantation spécifique

Type 2 : Composants réutilisables de bancs d'essai à travers différent implantations

### Composants de bancs d'essai réutilisables dans un même design

Dans le premier type, les composants de design peuvent être réutilisés à différents endroits d'un cycle de design, du niveau des blocs au niveau système.

La figure 1 montre un bloc (DUT) qui est intégré à un système ou un autre bloc. Ce bloc utilise certainement une portion qui génère des signaux, et cette portion de bancs d'essai peut facilement être réutilisée lors d'une intégration du DUT à d'autres blocs.

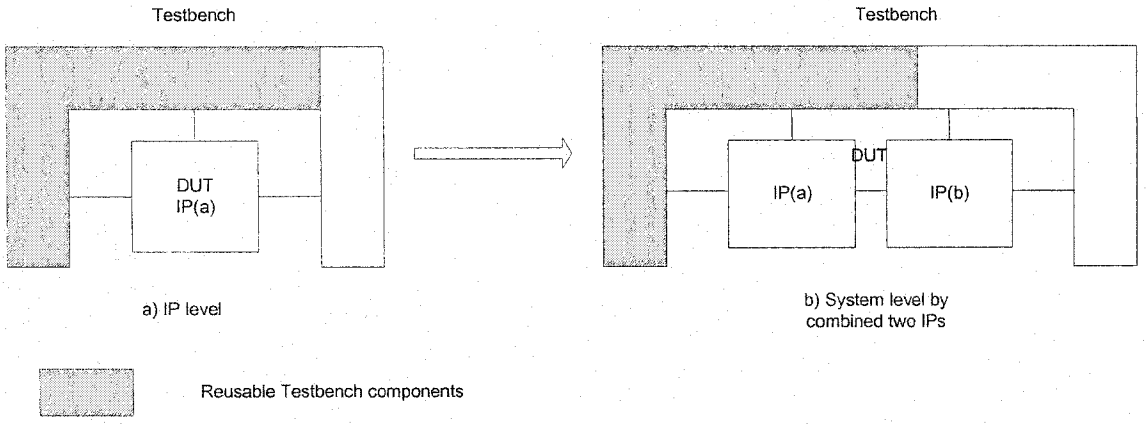



Figure 1: Réutilisation de composants de bancs d'essai d'un niveau bloc à un niveau système

**Composants de bancs d'essai réutilisables à travers différents designs**

Pour les besoins de la réutilisation dans les designs, plusieurs normes ont été définies afin de faciliter le travail d'intégration, dans PCI par exemple. Plusieurs compagnies produisent des composants compatibles à PCI, par exemple, Xilinx, Compaq, IBM, etc. Sur tous ces composants, la fonctionnalité PCI est la même et chaque banc d'essai qui voudra valider la partie PCI effectuera la même chose. Il n'est donc pas nécessaire que plusieurs équipes recréent leurs propres bancs d'essai et de réinventer la roue!

La figure 2 montre des composants de bancs d'essai implantés pour un design mais réutilisés dans d'autres implantations.

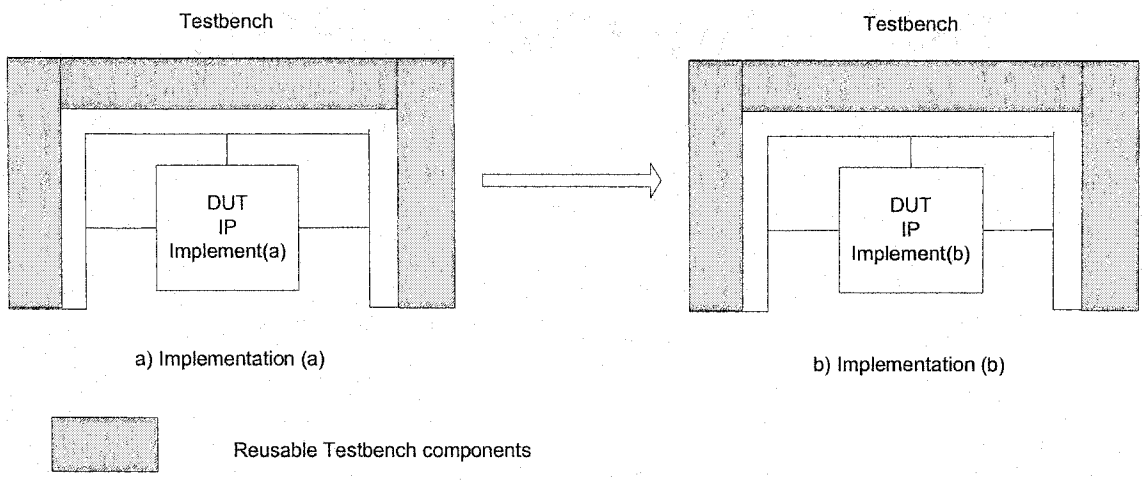



Figure 2: Réutilisation de composants réutilisés dans différents designs

### 3. Abstraction fonctionnelle

Pour réutiliser différents bancs d'essai à travers plusieurs designs, il faut trouver le comportement général à valider, de façon à retrouver ce que nous pourrons réutiliser pour chaque design. Le type ou le caractère des *testbenchs* qu'on pourrait réutiliser dépend fortement du niveau d'abstraction qu'on choisit.

#### Les niveaux d'abstraction fonctionnelle

Normalement, on retrouve des bancs d'essai bâtis au niveau RTL. Ce niveau, quoique très utilisé pour l'implantation de bancs d'essai, rend presque impossible la réutilisation de bancs d'essai à travers différents designs. Le niveau d'abstraction n'est pas acceptable et c'est pourquoi nous devons l'augmenter.

D'abord, nous débutons avec la spécification. Avec la même spécification, les fabricants peuvent implanter différents produits. Nous pourrions utiliser la spécification pour élever le niveau d'abstraction de ces produits. Reprenons PCI en exemple. Nous avons mentionné que IBM, Xilinx et d'autres implantent leurs composants PCI en se basant sur une spécification proposée par *PCI Special Interest Group* (PCI-SIG). Comme le montre la figure 3, les bancs d'essai générés se doivent d'être fonctionnels. À nouveau,, pour IBM, Xilinx, etc., les bancs d'essai générés sont valides pour tous les produits PCI et n'importe quelle équipe pourra par le fait même utiliser les bancs d'essai générés.

Comme la figure 4 le démontre, nous donc pouvons extraire les caractéristiques communes pour les protocoles de bus PCI, USB et AMBA. Ces protocoles de bus sont ainsi catégorisés en un seul groupe. En étudiant ce groupe, nous pouvons faire ressortir toutes les caractéristiques relatives au domaine des bus, puis générer des bancs d'essai fonctionnel à haut niveau d'abstraction valide pour tous ces protocoles. C'est ainsi que nous nous devons d'élever l'abstraction de la fonctionnalité à un niveau encore plus abstrait. Nous pouvons répéter ce processus pour atteindre un niveau d'abstraction suffisant pour étudier et valider à haut niveau les caractéristiques d'un groupe en particulier.

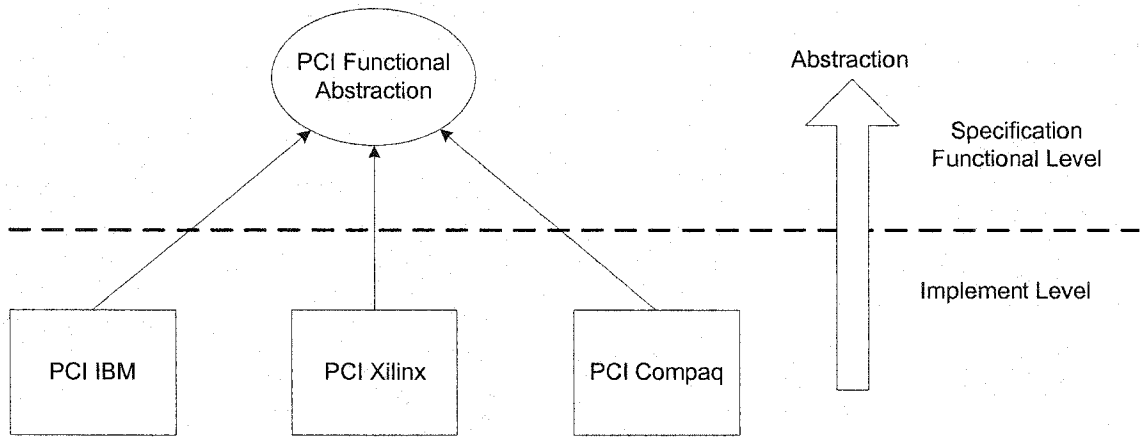



Figure 3: Niveaux de spécification fonctionnelle

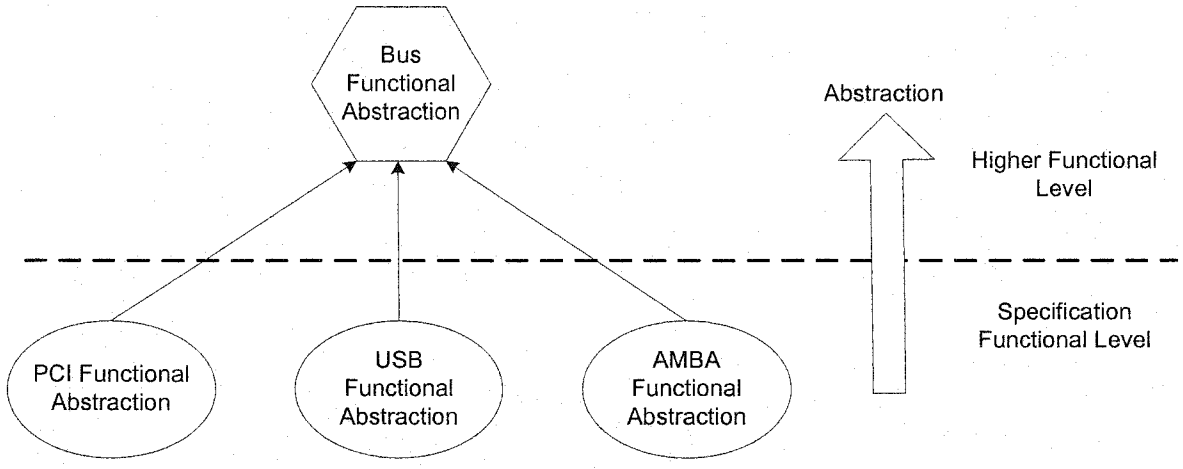



Figure 4: Niveaux d'abstraction élevés pour des bus

La figure 5 montre un schéma global de l'abstraction fonctionnelle. Plus le niveau d'abstraction est élevé, plus la fonctionnalité utilisée est générique et elle peut donc être utilisée plus facilement à travers différents designs. Par contre, plus le niveau

d'abstraction utilisé est bas, plus les caractéristiques implantées sont spécifiques à un design propre.

Grâce à cet arbre d'abstractions fonctionnelles, nous pouvons bâtir des bancs d'essai à différents niveaux d'abstraction, tout fonctionnels, qui visent certains niveaux spécifiques pour la réutilisation de bancs d'essai.

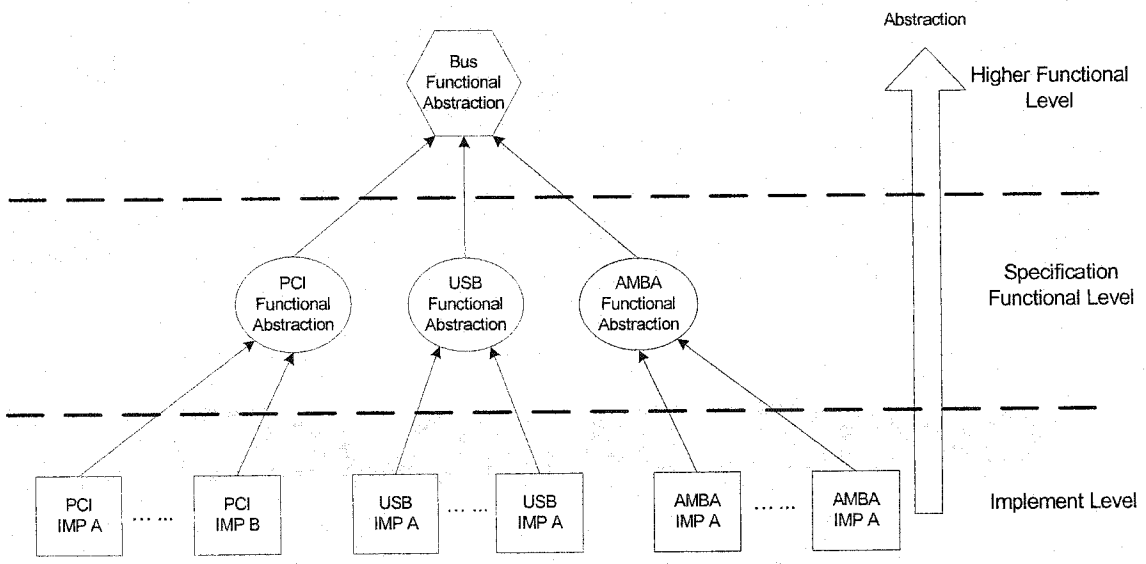



Figure 5: Schéma global de l'abstraction fonctionnelle

## 4. Une méthode pour l'abstraction fonctionnelle

Nous avons déjà discuté de l'abstraction fonctionnelle des systèmes dans la section précédente. Dans notre méthodologie, nous avons besoin d'un langage pour capturer cette abstraction. Ce langage devrait posséder les deux fonctions principales suivantes. Premièrement, il devrait pouvoir capter la fonctionnalité du système : il devrait posséder

toutes les caractéristiques pour permettre de décrire le système au complet en tant que spécification. Par la suite, l'abstraction pourrait y être intégrée. Deuxièmement, le langage devrait servir de fichier d'entrée standard pour notre outil de génération de bancs d'essai, qui est un de nos objectifs principaux. Les fichiers d'entrée pour l'outil de génération consistent au code des spécifications fonctionnelles et ces fichiers sont évidemment nécessaires pour générer des bancs d'essai. Cette information provient des spécifications du système.

**5. Méthodologie de réutilisation de bancs d'essai**

Nous introduisons ici la méthodologie qu permet de construire des bancs d'essai au niveau fonctionnel. En premier lieu, nous nous concentrons sur le premier niveau d'abstraction, c'est à dire le niveau des spécifications du système. D'abord, nous devons capturer les spécifications fonctionnelles du système à l'aide de SDL. Une fois la spécification vérifiée, l'équipe de développement peut se diviser en deux groupes : un groupe traduira la spécification SDL en RTL et l'autre groupe, celui de la vérification, construira les bancs d'essai. Comme ce travail est exécuté en parallèle, le temps de développement est grandement réduit comparativement à la méthode actuelle, qui consiste à attendre que les spécifications RTL soient prêtes avant de passer à l'écriture des bancs d'essai. Il est important de mentionner que les bancs d'essai qui développer à ce stade n'ont pas beaucoup de détails d'implantation, simplement parce que l'information provient d'une description SDL de haut niveau. On désire plutôt que les

bancs d'essai soit réutilisable pour n'importe quelle implantation, donc qu'il soit générique.

Au cours de la génération des bancs d'essai, la méthodologie prévoit que le banc d'essai soit séparé dans des modules selon ce qui a été proposé dans la méthodologie de partitionnement par aspects. Ceci permet aux bancs d'essai de type 1 (section 2) d'être réutilisables, de même que cela ouvre la porte à l'exploration et l'intégration de techniques orientées objets au cours de travaux futurs.

Avant d'utiliser les bancs d'essai pour simuler les DUT, nous devons associer le banc d'essai à un design spécifique de plus bas niveau, qui comporte plusieurs signaux à définir.

**6. Outil de génération : *Reusable Testbench Generation Tool (RTGT)***

Nous implantons notre méthodologie dans un outil de génération automatique de bancs d'essai. Cette approche permet de réduire énormément le temps de vérification des designs.

Tel qu'on peut voir à la figure 6, l'ensemble d'outils proposé comporte 4 sous-ensembles. Premièrement, un outil SDL qui traduit les spécifications graphiques dans une notation textuelle. Pour cela, nous utilisons l'outil commercial Telelogic. Ensuite, nous avons un analyseur syntaxique (*parser*) pour SDL afin de lire les spécifications textuelles du

système. Ensuite, une interface graphique permet à l'usager d'interagir et le noyau de l'ensemble d'outils : *Automated Testbench Generator* (ATG). La sortie générée constitue un composant en langage de vérification *e*.

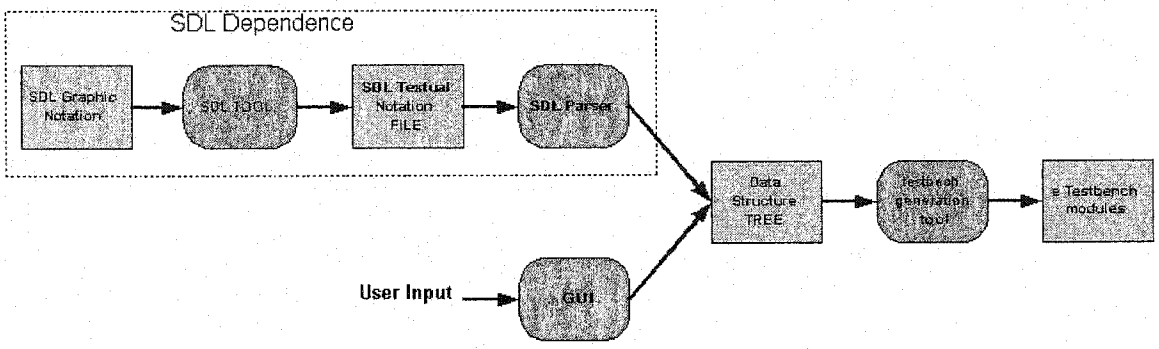



Figure 6: L'ensemble des outils RTGT

## 7. Conclusion

Dans ce projet, nous avons d'abord exploré une méthode qui nous a permis d'obtenir une réutilisation de bancs d'essai à travers différents designs. Nous avons également présenté et expliqué l'abstraction fonctionnelle de même que son rôle dans notre méthodologie. Nous avons aussi classifié tous les niveaux d'abstraction. Pour la première abstraction (niveau de spécifications fonctionnelles), nous avons choisi SDL pour capturer la fonctionnalité du système. Enfin, nous présentons une méthodologie pour construire des bancs d'essai réutilisables intégré à un outil de génération automatique – RTGT.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OFTABLES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| AMBA | Advanced Microcontroller Bus Architecture |
| APHVR | Aspect Partitioning for Hardware Verification Reuse |
| ASIC | Application Specific Integrated Circuit |
| ATG | Automated Testbench Generator |
| DUT | Design Under Test |
| EFSM | Extended Finite State Machine |
| FA | Functional Abstraction |
| HAFL | Higher Abstraction Functional Level |
| HDL | Hardware Description Language |
| HVL | Hardware Verification Languages |
| IP | Intellectual Property |
| OOP | Object-Oriented Programming |
| PCI | Peripheral Component Interconnection |
| RTGT | Reusable Testbench Generation Tool set |
| RTL | Register Transfer Level |
| SDL | Specification Description Language |
| SFL | Specification Functional Level |
| SoC | System-on-Chip |
| USB | Universal Serial Bus |
| UML | Unified Modeling Language |

# LIST OF APPENDIX

# CHAPTER 1
# INTRODUCTION

## 1.1 MOTIVATION

It is generally recognized that functional verification is the most time consuming phase in the design cycle. With the development of high-density ASIC and SoC designs, verification tasks are facing a new challenge due to the complexity of integrating a large number of functional units. Complicated testbenches and test scenarios have to be defined to verify large multiple sub-systems and the interaction among them. Special techniques need to be developed in order to verify integration of various SoC components.

According to a recent survey conducted by our research group [2], the development of complex testbenches requires more than 50% of the overall hardware design for no less than 85% of respondents. Moreover, only 23% of our respondents used pre-defined frameworks to build testbenches and only 35% follow an internal testbench modules reuse policy.

New verification languages and tools have been developed and are well used in the process of project design, for example, VERA [11] and Specman [9]. However, those efforts have proved to be only partially effective [7]. Languages and tools play an important role, but a largely secondary one, in the functional verification process [7]. The real key for the development of a more efficient approach to system-level functional verification lies in a new methodology [7]. To reduce the time used for testbench development, reuse is one solution that can significantly accelerate verification cycle. Reuse of design components has already played a very important role in complex system development. Then, developing a new and efficient methodology aimed to verification reuse becomes our main goal in this project.

Our research group has already presented a methodology -- *Aspect Partitioning* [8], which promotes hardware verification reuse. This methodology efficiently constructs a verification test environment with an object-oriented design technique. Testbench modules being built by this method are reusable and extensible. This method focuses on software reuse. Testbench built for one IP can be reused when verification is rising from block level to system level or this IP is used in another system. However, testbench can hardly be reused by the other IP implementations. We are seeking a methodology to reuse testbench not only based on one IP, but also between different IPs.

In order to be reused among different IP implementations, the testbench components must be general enough to be adopted by different designs. In other words, we need to capture common features of the *Design Under Test* (DUT), and the testbench components being built with these common features will be general to these DUT. The information needed to build testbenches for a specific design is the functionality and structure of the design. Obviously, it is not a good way to retrieve this information from *Register Transfer Level* (RTL), because the RTL implementation varies from project to project. Although their functions may be the same, different projects would have different implementations, which make it hard to retrieve information standing for common functionality from implementations. A proper way to capture functionality comes from high-level description – specification. Because all implementations are based on the same specification, abstract functionalities that are captured from specifications are common for those implementations. We call this method that captures general functionalities among certain design implementations *Functional Abstraction* (FA). Consequently, testbench components being built with abstract functionality are reusable.

The constraint of this FA method is that testbench models are reusable only within the same design family (designed with the same specifications). To overcome this, we should raise the abstraction level to cover more implementations.

At functional verification stage, coding testbench is an important and time-consuming work. A good testbench can accurately emulate system design environment and offer stimulus that can achieve high coverage. HDL languages, such as VHDL or Verilog are normally used to writing testbenches, however, they are not ideal for verification tasks because they lack advanced temporal logic or complex data structures. The use of *Hardware Verification Languages* (HVL), such as Verisity's *e* [9], improves the verification process by addressing the requirements of complex functional verification software development [8].

High-level data structures or object-oriented features make verification languages more efficient languages to develop testbenches than VHDL and Verilog. However, a new language brings additional training to verification engineers who normally have good experience of HDL languages. That is the reason that HDL languages are still the most widely used languages to code testbenches.

To solve this problem, automatic testbench generation is a possible solution. If a tool could automate the generation of most testbench components, it would dramatically reduce the burden of verification engineers. In our project, we implement a testbench reuse methodology using a tool that generates testbench components. The input of this tool is design specifications and its outputs are testbench components coded with a HVL language, the *e* language.

To standardize inputs to the tool, we need a standard textual specification as input that could be recognized by the tool. As a system functionality modeling language, the *Specification Description Language* (SDL) is chosen for its graphical and textual notation features. SDL graphical presentation makes it easy to describe the system and textual notation can work as standard inputs for the tool.

This tool is developed with the *rule-based* technique. The rule here tells what information will be needed to build a testbench and how to process this information. Each rule is related to a software module of the tool, and the different combinations of rules are applied to different DUT features. That makes the tool flexible enough to cover different kinds of DUT requirements. The *Rule-based* technique also brings advantages for the expansion of the tool in the future. The designer must add, modify or change relative rules, and the software modules accordingly, in order to update the tool. No modification is required in other parts of the program.

## 1.2 RESEARCH GOALS

The primary objectives of this master thesis are to:

- Present a methodology to capture system functional abstraction. SDL is used in this project to describe a system, but the methodology can be adapted to support any language for system design. The use of specifications for verification will be advocated in this methodology.

- Present a methodology to build reusable testbenches. This methodology includes two kinds of testbench reuse: from block level to system level; and among different implementations with the same specification.

- Utilize *rule-based* techniques to develop a *Reusable Testbench Generation Tool set* (RTGT). This help automating the generation of testbench based on a reusable testbench methodology. Rules help taking SDL system description as input and generating a testbench as output. The output testbench is written with the Specman's language *e*. *Rule-based* technique makes RTGT flexible and expandable.

- Implement rules in a RTGT. Standard C++ is used to develop this tool and make it portable for Unix and Windows platforms. Tcl/Tk is used to develop the graphical user interface of this tool.

## 1.3 THESIS ORGANIZATION

Following this chapter, Chapter 2 provides an overview of functional verification. In this chapter, we briefly introduce formal verification and present the key components of functional verification in detail. HVL and language *e* will also be introduced.

Chapter 3 focuses on reusable testbenches. In this chapter, we introduce current approaches and analyze their advantages and limitations. Also, we present additional ideas to explore a new method to increase testbench reusability. *Functional abstraction* will be discussed in this chapter as a means to increase reusability through raising the abstraction level of design.

Chapter 4 introduces a method to model system functionality and present a methodology to build reusable testbenches. In this chapter we will introduce three languages that can be used as standard modeling languages: SDL, *Unified Modeling Language* (UML) and SystemC. First, comparisons will be made among these three languages. Secondly, the choice of SDL as an executable specification language in this project will be motivated. Adapting the method to describe system functionality with SDL, we developed a methodology to build functional reusable testbenches. A concrete example is used to illustrate the methodology. This methodology will be implemented in a RTGT that automates testbench generation.

Chapter 5 presented the implementation of the RTGT that supports the methodology presented in Chapter 4. *Rule-based* techniques are used to develop this tool. The nature of rules, why they are used and how they are implemented are discussed in this chapter.

Finally, in Chapter 6, the conclusion from this project are presented and future works are proposed.

# CHAPTER 2
# OVERVIEW OF FUNCTIONAL VERIFICATION

## 2.1 INTRODUCTION

In today's micro-electronic systems, gate counts and system complexity grow exponentially, along with the challenge of hardware verification. It is impossible to assume that a system works well just because it is designed and implemented carefully. Also, a slight change from the last version of an implementation can cause potential functional error. Every IP being verified independently, there is no guarantee that the system will execute correctly at system integration.

The functional verification phase is generally recognized to be the most important barrier in rapid design cycle of product development. The designer must of course construct an implementation that fulfills desired functionality, but a difficult challenge is to construct an implementation that meet the time-to-market constraint. Introducing a system to the marketplace early can make a big difference in the system's profitability, since market windows for products are becoming quite short, with such windows often measured in months. In some cases, each day that a product is delayed from introduction to the market can translate to a one-million-dollar loss [37]. The difficulties of meeting the time-to-market constraints are growing due to increasing complexity. More functionality added into a system increases the pressure on verification engineers. No exiting method can tell when a design is efficiently verified. Normally, the verificaiton phase can last as long as the developers are willing to invest on it. That is the reason why industry puts so much effort on verification.

Both tools and methodologies were and are still being developed to assist verification engineers to deal with rapid and accurate verification requirements. However, according

to a recent survey conducted by our research group [2], the majority (50% ~ 69%) of the respondents do not have a design flow for verification methodology and some companies even do not have a plan for verification. The production of testbenches is a very consuming task that requires more than 50% of the overall hardware design time for no less than 85% of the respondents. All these data show the importance of verification methodologies and tools development.

In this chapter, we will give an overview of verification. Some key words will be defined and current approaches to verification will be presented. Also, we will raise problems associated with current solutions.

## 2.2 TERMINOLOGY: VERIFICATION, FORMAL VERIFICATION AND FUNCTIONAL VERIFICATION

### 2.2.1 Verification

Verification is a process used to demonstrate the functional correctness of a design [1]. According to the difference among verified items, verification is divided into formal verification and functional verification.

### 2.2.2 Formal Verification

We will not discuss formal verification in detail in this thesis. Readers, interested in this field, could refer to relevant documentation [12] [13].

In today's industry, there are two main kinds of formal verification tools used. They are called Property Checker or Model Checker and Equivalence Checker.

**Property checker**

Given a specification, it is first converted into some kind of a finite state machine representation. All these state machines could be verified to find isolated states or some dead path. For example, given some kind of interface design, it is relevant to find out whether any bus conflict will ever occur, or that every request signal will be followed by an acknowledge signal.

**Equivalence checking**

In order to perform equivalence checking, model under verification and reference model are fed to the tool and the tool determines whether the designs are equivalent or not. This formal verification process mathematically proves that the outputs are logically equivalent, and that the transformation preserved its functionality [1].

Equivalence checkers are classified as either combinational or sequential verifiers. Combinational equivalence tools are capable of validating the equivalence between two large designs that have the same number of states. Sequential equivalence tools, on the other hand, can take two designs and verify the input/output behavior between two designs at any level of abstraction [13].

### 2.2.3 Functional Verification

The main purpose of functional verification is to ensure that a design implements the intended functionality [1]. As showed in the Figure 2.1, functional verification verifies whether or not the specification is correctly coded with an HDL language.

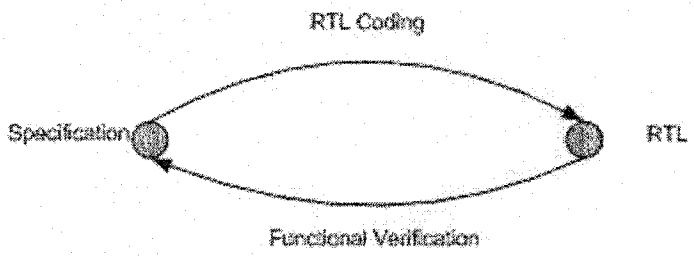



Figure 2.1: Functional verification paths

Every design starts with a specification. Very often, specifications are written using a natural language. Then, the problem is whether or not the interpretation of specification is equivalent to the design intent. It is impractical to ensure that the specification is totally correct and implemented exactly as the specification defines. Even very experienced engineers cannot avoid making mistakes. As errors are inevitable, some method must be applied to check these works. Functional verification is the method utilized in this phase.

Then, when the specification is implemented in RTL, there are two potential points that may introduce functional errors. One is whether or not the specification is accurately documented according to the design intent. The other is, during the process of implementing the specification into RTL, most of the time, we cannot guarantee engineers understand specifications correctly or implement RTL with HDL language without bugs.

Functional verification problems get even worse for *System-on-Chip* (SoC) designs. SoCs may contain the following components: a processor or processor sub-system, a processor bus, a peripheral bus, and many peripheral devices. And all these components are in one chip. Because SoCs are implemented using reusable IP cores, besides the complexity of verifying millions of gates ASIC, SoC add their own sets of new problems. Whenever there is a functional error, verification engineer must point out those errors caused by IP core or existing errors during system integration. Obviously, it is a big challenge considering multi-million gates, complex architecture systems.

In a complex SoC design flow, functional verification is very important: any behavioral or functional bug escaping from this phase will not be detected in the subsequent implementation phases, and will surface only after the first silicon is integrated into the target system, resulting in costly design and silicon iterations [16]. Therefore are functional errors should be detected as early as possible during the development phases. The earlier the errors are found, the less they could cause damage and the easier they are to solve. For example, if an error in an IP core has not be exposed during IP development, and it causes a system to function incorrectly after being integrated, much more efforts will have to be spent to find out where the error exists.

Functional verification is mainly based on simulation. To simulate the behavior of a hardware component, we need to provide vectors with input stimulus and response. A testbench is normally developed to handle this.

## 2.3 THE KEY COMPONENTS NEEDED TO PERFORM FUNCTIONAL VERIFICATION

Standard techniques in functional verification include hardware modeling and simulation, stimuli generation, coverage analysis, and developing or using checker to compare implementation to reference models.

To perform functional verification, the necessary components are: a DUT, a testbench, testcases, and a test generator. In addition, a verification plan, self-checking machines, and a coverage model are all playing very important roles during functional verification.

### 2.3.1 Testbench

The term "testbench" usually refers to the code used to create a pre-determined input sequence to a design, and optionally, to observe the response [1]. It is commonly

implemented by using VHDL or Verilog, but with the development of some HVL, testbench is more likely built by object-oriented languages.

The idea of a testbench is to set up an easy way to verify a DUT without typing simulation commands into the simulator every time we make a change in the HDL code. Testbenches emulate a hardware breadboard into which the developers "install" a synthesizable design description for the purpose of verification. Testbenches can be quite simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file and writing test results to the screen and to a report file. A comprehensive testbench can, in fact, be more complex and lengthy (and take longer to develop) than the synthesizable circuit being tested [20].

Figure 2.2 shows how a testbench interacts with a DUT. Testbench emulates system environment of design, provides input vectors to stimulate the DUT, and consequently monitors its outputs. The challenge of testbench generation is how to offer efficient stimulus that can achieve high coverage of design's function and self-check its outputs with what is expected

.

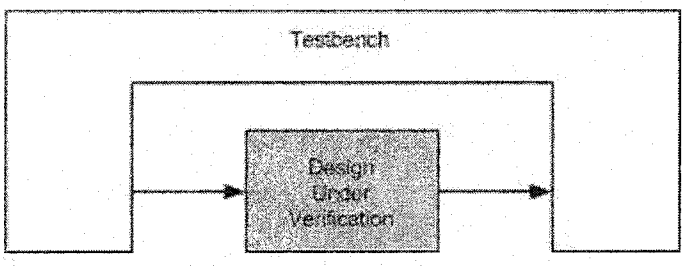



Figure 2.2: Structure of a testbench and design under verification

### 2.3.2 Testcases

Testcases are the sequence of fixed input and expected output values to stimulate the design. This sequence of values could be described by using multi-dimensional arrays or arrays of records.

### 2.3.3 Test generator

A test generator is a module that generates testcases. The test methods will have different influence of the efficiency and simulation time. We will discuss this issue in the following section.

**The DUT input verification space**

One way to define the set of input combinations for a given device is as an n-dimensional space of $2^n$ combinations of discrete points. Where *n* represents the number of DUT input pins at each dimension. Each of these points has two values: one and zero. Adding the time domain (because a test is typically a sequence of combinations) results in an n+1 dimensional space. This space is the DUT input verification space in which stimuli can be generated [21].

Figure 2.3 [21] shows an example of such a DUT verification space. The example illustrates the possible inputs for a sequence of packets, each with two fields: length and type. During functional testing, many sequences of packets are generated, each containing different attribute values (length, type, and time). Each black dot represents an assigned specific value. A set of dots connected by a broken line represents a test. Finding the bugs consists of generating input sequences that reveal the bugs in the DUT behavior [21]. The bugs in this figure are represented as black bombs.

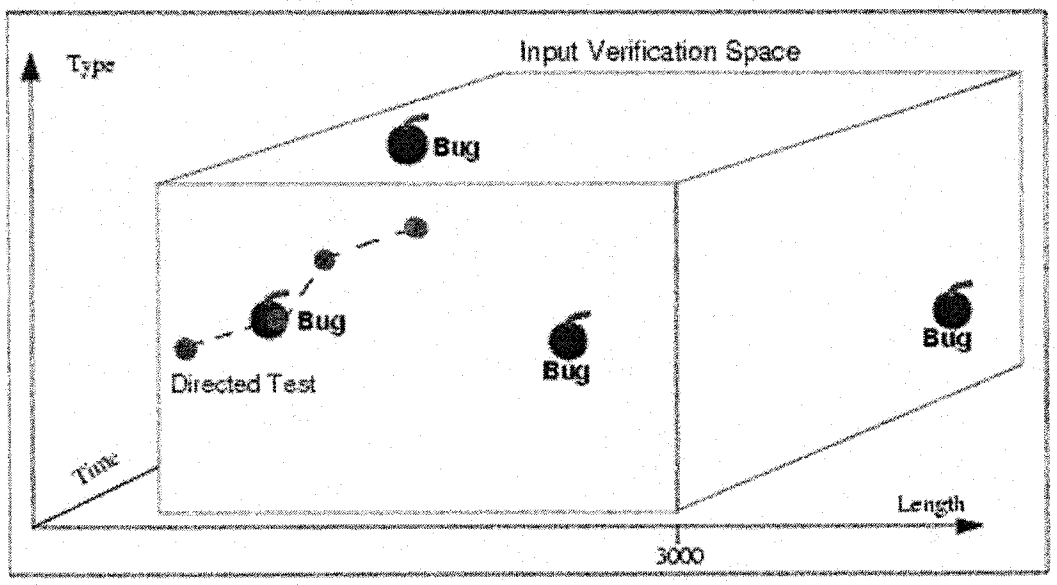



Figure 2.3: Directed test, represented by the set of black dots. The black area represents the space that theoretically should be covered by testing. [21]

Tests may be classified as: exhaustive tests, directed tests, random tests, and directed-random tests.

**Exhaustive tests** stimulate the entire set of combinations of inputs scenarios. Considering long simulation time, it is usually impossible.

**Directed tests** target specific aspects of the DUT behavior. These tests are represented as sets of black dots in the figure. The advantage of directed tests is that ensure specific features of interest will be tested, however; on the other hand, it is hard to develop a directed test to cover huge input space.

**Random tests** randomly select combinations within the given space. It takes only little effort to cover a huge space and to generate combinations that verification engineer probably have not considered. The disadvantage is that it may generate lots of uninteresting test cases and this method causes longer simulation times.

**Directed-random testing** provides a way to create directed or random tests by using constraints, which a user define to target a subspace of interest in the DUT input verification space. For example, Specman Elite's generator randomly generates input sequences in the defined verification space. By running the same directed-random test many times, the user get different combinations within the same test space. Figure 2.4 [21] shows two test runs, run1 and run2, designated by black broken lines.

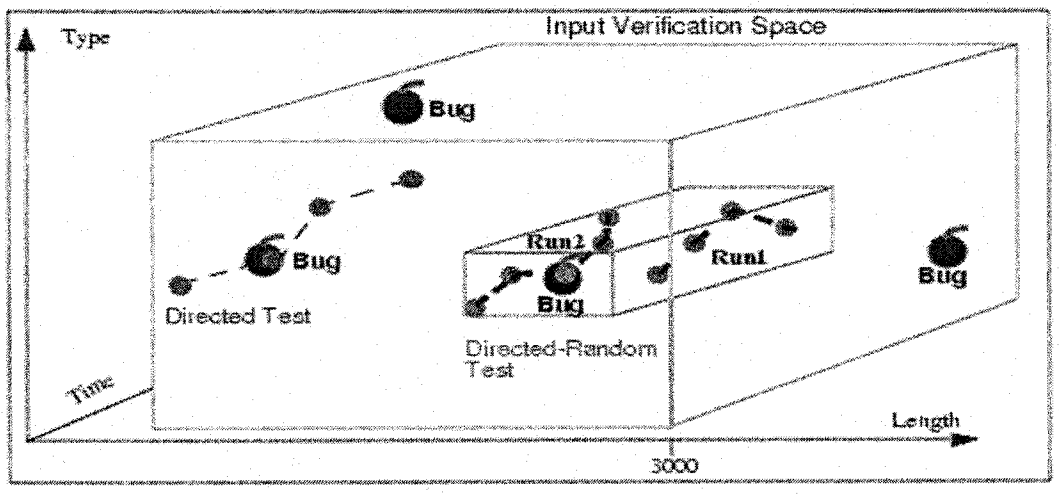



Figure 2.4: Directed-random testing [21]

### 2.3.4 Verification plan

Like for a design project, a specification is generated to define detailed functionality of a system. The verification plan specify which functionality is to be verified during the verification phase. Detailed schedules should be produced for the verification. Also, the plan should define what needs to be verified and when the verification will be completed to the required degree of confidence.

A verification plan should define which features must be exercised, under what conditions and what are the expected responses. All verification engineers in a team must

have a clear idea of how many testbenches need to be written, how complex they need to be, and how they depend on each other.

Verification plan can also tell the level of granularity for the verification effort, IP-level, unit-level or system level and what types of testcases need to be generated for each level: black-box, white-box, or grey-box verification (see Functional Verification Approaches in Section 2.4).

Also, the level of abstraction can be decided from verification plan. Higher level of abstraction do not provide detailed timing control. If they are required, verification should proceed at lower levels of abstraction.

Verification teams can benefit a lot from a clearly defined verification plan. However, the importance of a verification plan has apparently not received enough attention in the industry. From the survey held by our research group [2], some companies do not make verification plans.

### 2.3.5 Self-checking Testbenches

For a same design, designer can examise output manually by comparing them to their expected value. However, for systems, simulations can last thousands of clock cycles, while stimulating hundreds of inputs that cause hundreds of outputs to be verified. This causes a significant burden on verification engineers to compare every data hoping not to miss any error. Also, this process need be done for every simulation run of every testbench. This motivate developing methods that could compare the simulation results with the expected outputs automatically. Such method is thus called self-checking.

Self-checking means that testbenches verify themselves. Checkers are used to verify results at run-time by modeling the expected response at the same time as the stimulus. Checking can be classified into: Data Checking and Protocol Checking.

**Data checking**

Data checking is based on comparing the output with the input. To match output to input, we need to build a *scoreboard*.

One of the basic concerns when checking data is to verify whether the output data items collected from the DUT match the corresponding data items injected into the DUT. Scoreboard records each inject data item and note when a matching data item is found. Figure 2.5 illustrates an example of a scoreboard during a sample run. This example comes from Specman online document [21].

With scoreboard checking, we verify that:

- Every input has a matching output
- Every output has a matching input
- Output conforms to expected results for the inputs

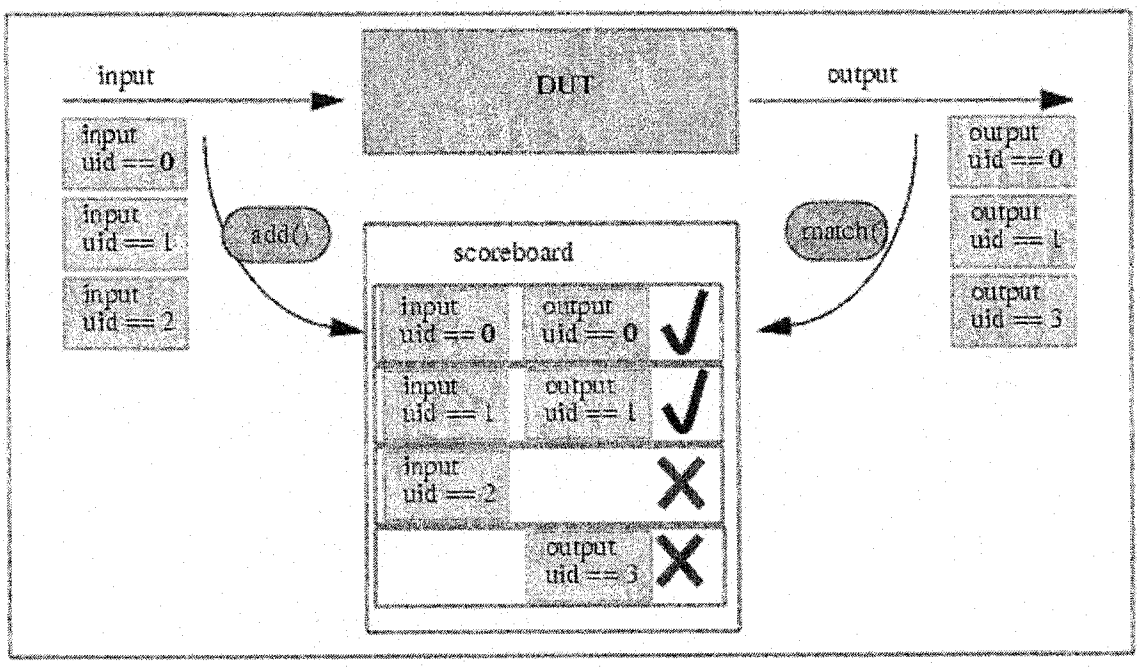



Figure2.5: Scoreboard status after injecting 3 inputs and collecting 3 outputs [21]

**Protocol checking**

Protocol checking consists in verifying if a DUT respects a protocol. A protocol is a set of rules that determine the timing and order of appearance of data and events. To perform protocol checking, we need to develop a protocol checker. Within the checker, protocol rules being verified must be well defined. Figure 2.6 is an example from IBM PCI-X bus protocol checker.

Figure 2.7 shows a PCI checker implemented with Verisity's verification tool Specman Elite [9]. This example comes from Specman online documentation [21]. The checker validates selected rules taken from the PCI specification (revision 2.2). It is supposed to be used in an environment with a PCI agent (either as the entire DUT or as part of a larger system).

| # | Protocol Rules & Description | Reference of PCIX Spec: | Error Code |
|---|---|---|---|
| 0 | An initiator must drive the address for 4 cycles before asserting frame on a configuration access. | PCIX 1.0, p.57, sec.2.7.21 | XMSG0 |
| 1 | In a burst with less than 4 data phases, the initiator must deassert FRAME two clocks after the target asserts TRDY | PCIX 1.0, p.87, sec.2.11.1.1 | XMSG1 |

Figure 2.6: IBM PCI-X Bus Protocol Checker [40]

The checker consists of *expect* constructs, each using a *temporal* expression to monitor the bus operation and identify incorrect behavior. The checker also uses *temporal* expressions to collect coverage information about the PCI agent operation. *Expect* is used to define a behavioral rule and a *temporal* expression is a combination of events and temporal operators that describes behavior. For more information on *expect* and *temporal*, please refer to Specman online documents [21].

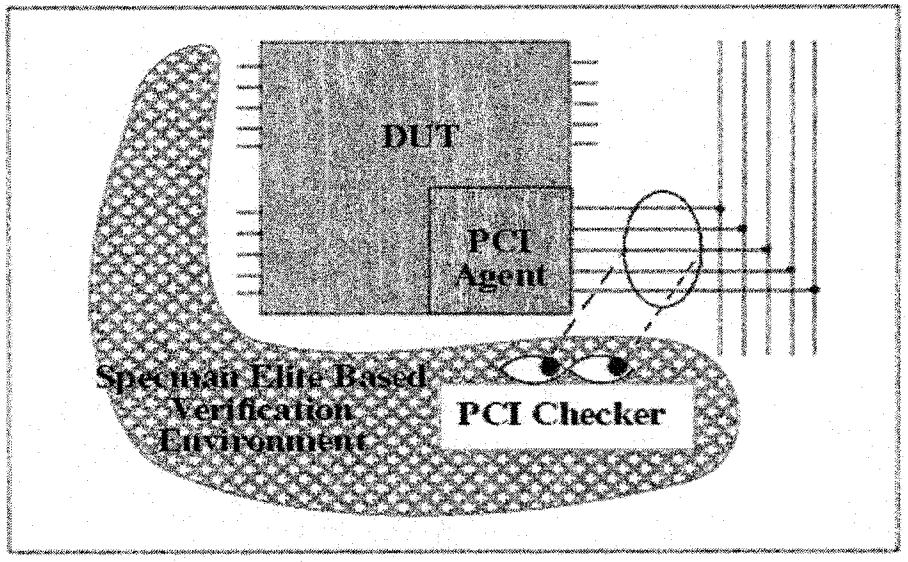



Figure2.7: Typical Checker Environment [21]

### 2.3.6 Coverage

During functional verification, verification engineers hope to test all the functionality and code of a DUT. However, this goal is usually not achieved by only spending more time on verification. If a designer did not consider all the scenarios during test plan development, and has no method to remind him during verification, time will likely be wasted on repeated testing of the same functionality of a DUT or the same parts of the code.

Coverage metrics is used to help the verification engineer have a clear idea about how well the verification is progressing. Coverage metrics should be used at the early stages and throughout the verification process to quantify how well the functionality has been covered so far. That will save significant human and machine resources, shorten time-to-market and eventually contribute to raise the quality of a product.

Coverage can be classified into code coverage and functional coverage.

**Code coverage**

Code coverage reflects how thoroughly the HDL code was exercised. A code coverage tool traces code execution, usually by instructing or modifying the HDL code.

Code coverage tool includes line/block coverage, arc coverage for state machines, expression coverage, event coverage and toggle coverage.

**Functional coverage**

Functional coverage perceives the design from a user's or a system point of view. Functional coverage tools quantify the cover of typical scenarios, error cases, corner

cases, protocols, and some combination of executions. For instance, a situation for which an interrupt happens at the same time as a state transaction, or the injection of two erroneous packets in a row [25].

## 2.4 FUNCTIONAL VERIFICATION APPROACHES

Functional verification can be accomplished by using three complementary but different approaches: black box testing, white-box testing, and grey-box testing.

### Black-Box verification

With a black-box approach, functional verification must be performed without any knowledge of the actual implementation of a design. All verification must be accomplished through available interfaces, without directly accessing the internal state of the design, and without knowledge of its structure and implementation. This method suffers from an obvious lack of observability and controllability. It is often difficult to set up an interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem. This difficulty arises from the frequently long delays between the occurrence of a problem and the apparition of its symptom on the design's outputs [1].

The advantage of black-box verification is that it does not depend on any specific implementation. Whether the design is implemented in a single ASIC, multiple FPGAs, a circuit board, or entirely in software, is irrelevant. A black-box functional verification approach forms a true conformance verification that can be used to show that a particular design implements the intent of a specification regardless of its implementation [1].

The black-box approach is the only one that can be used if the functional verification is to be implemented in parallel with the implementation of the design itself. Indeed, there is no implementation to guide verification program development at that stage [1].

Black-box verification is often used in early stages of functional verification. After specifications have been completed, development groups can be divided into two teams: a design team that implements the specification at RTL level with a HDL language, and the verification team that builds testbenches. Because of lack of detailed implementation information, verification engineers at this stage can only develop testcases with a black-box approach. Although these testcases are not oriented to specific design features, they are good enough to verify a design functionality according to its specification. Since these testcases are developed at the same time as the design team implements the specification, functional verification could be implemented while designers are still coding with HDL. During system functionality is being verified, design implementation information is ready to be used by verification engineers to develop testcases that aim specific design features. This kind of "pipeline" could reduce system development turn-around time.

**White-Box verification**

In contrast to black-box verification, a white-box has full visibility and controllability of the internal structure and implementation of the design being verified. This method has the advantages of being able to quickly set up an interesting combination of states and inputs, or isolate a particular function. It can then easily observe the results as the verification progresses and immediately report any discrepancies from the expected behavior [1].

However, this approach is tightly integrated with a particular implementation and can be difficult to use on alternative implementations or future reuse. It also requires detailed

knowledge of the design implementation to know which significant conditions to create and which results to observe [1].

With the white-box approach, efficient testcases could be generated to target specific scenarios, such as to 'push' when a stack is already full. White-box verification appears to be efficient than black-box verification, however, as mentioned above, white-box testcases are hard to reuse and their development and use require verification engineers to fully understand a design implementation, which makes IP cores to be used in different systems harder to use. Thus, white-box verification is a useful complement to black-box verification, but it cannot replace it. According to the survey held by our research group [2], majority (50%~69%) of respondents use a white-box verification approach at unit and integration verification levels and use a black-box verification approach at a system verification level.

**Gray-Box verification**

Grey-box verification is a compromise between black-box verification and white-box verification. The former may not fully exercise all parts of a design, while the latter is not portable [1].

As in black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However, the particular verification being accomplished is intended to exercise significant features specifical to the implementation.

Grey-box allows testbench reuse, especially for designs within the same family (different implementation with same specification). For example, consider the PCI interface. Different vendor's product may have different specific features. Gray-box testbench could be used to verify functionality for all PCI implementations, while white-box verification could be used to verify specific feature of each one.

## 2.5 HARDWARE VERIFICATION LANGUAGE (HVL)

As we know, in the software domain, assembly-level language is gradually replaced by high-level language C or C++ for the feature of high-level data structures or object-oriented programming. These features raise the level of abstraction and make software reusable.

VHDL and Verilog are very often used to build testbenches. Both of them focus on low-level hardware description and are not very good to support high-level data structures and object-oriented features.

In the verification domain, the emerging market for reusable testbench components highlights the need for high level, adaptable software for testbenches [27]. With the emergence of SoC, new challenges must be faced for functional verification. A SoC system is built by integrating IP cores that were already developed. Testbench for SoC system requires the ability of high-level integratation of these IP components. Low-level hardware description languages are not designed for these kinds of work. This provides an opportunity for languages specifically designed for verification to emerge in the market. These languages are called *hardware verification language*s (HVLs).

Nowadays, there are three popular HVL languages in the market: *e/Specman* from Verisity [9], *VERA* from Synopsys [11], and *TestBuilder* from Cadence [28]. Vera, like Verisity's *e,* is a specialized language that lets users build testbenches at a higher level of abstraction than Verilog or VHDL. This object-oriented language supports complex data structures, has built-in data types aimed at verification and lets users model the execution of events in time [36]. We will introduce *e* language in Section 2.5.1. In addition to VERA and *e,* Cadence has fielded TestBuilder. TestBuilder is an open-source class library that extends C/C++ to be used as an effective testbench language. It encapsulates hardware signal details and provides transaction-level abstraction to simplify the development of sophisticated tests [28].

These three languages face tough competition from each other, but no one has yet won the testbench language war. "If you are doing 100% verification I think Verisity's Specman can be useful. The verification language market suffers from the same problem as the "C" market, namely fragmentation and too many vendors pushing their own solution." Said by Anders Nordstrom of Nortel. One of our goals is to develop tools that automate the generation of testbenches with HVL, in order to relieve engineers from learning a new language. This work is part of a larger project supported by an industrial partner, PMC-Sierra. Because PMC-Sierra uses the *e* language as a functional verification language, it is also used in our project.

### 2.5.1 Overview of *e*/Specman

The *e* programming language enjoys widespread use in the microchip industry with applications to specification, modeling, testing and verification of hardware systems and their operation environments. Unique features of *e* include a combination of object-oriented and constraint-oriented mechanisms for the specification of data formats and interdependencies. They provide also interesting mechanisms of inheritance, and an efficient combination of interpreted and compiled code [26].

#### 2.5.1.1 The Aspect-Oriented Features of *e*

The basic function of *e* is to define structures, and then extend those structures in other files. A structure in e, just like a class in other object-oriented programming languages, such as C++, may declare fields and methods. Structures may also contain several unique declarative components, including constraints (affecting initial values assigned to fields), event definitions (for monitoring DUT behavior), and temporal properties (checking protocols, etc.)[26].

The following examples [26] drawn from a verification environment for a packet switching device, demonstrates the aspect-oriented features of *e*.

```
type packet.kind: [empty, short, long];
struct packet {
    i: int;
    j: int;
    kind:  packet.kind;
        keep i < j  + 1;
        keep j in [1..5];
};
```

The *struct* declares a structured object. The *keeps* are constraints that affect initial values assigned to the fields mentioned whenever an instance of this class is created – Specman resolves such constraints during a test run in order to generate a random, directed stream of data for the DUT.

The following example tries to add a few constraints on top of an existing environment. It uses inherited feature of language *e*.

```
// test1.e
extend packet {
        keep i == j;
        keep kind ! = empty;
};
```

The *extend* adds structure members to a previously defined *struct* or *struct* subtype, so that the extension only applies to the subtype.

In the following example, we modify a predefined method (one that has already been defined for *struct*)

```
extend packet {
        post.generate() is also {
            sys.packet_count += 1;
    };
};
```

The *is also* is used to extend a regular method, replace or extend the action block in the original method declaration with the specified action block.

### 2.5.2 Limitations of HVL

HVL languages bring software reuse ability to hardware testbench generation with its object-oriented programming features. However, using verification languages also comes with the trade-off of the requirement for additional training and tool costs.

HDL languages, such as VHDL, Verilog, are very popular in hardware design. With the tool's support verification team, as part of a develop team, there is no specific difficulty to build testbenches with one of these well known languages. That is the reason why most companies are still using HDL languages to build their testbenches, although HVLs are more suitable for verification than HDLs. The main advantage is that there is no need to pay additional languages/tools training and no need to pay for additional tools.

To promote these HVLs, new methodology must be developed to show to the industry the significant benefits HVLs can offer. If this new methodology and new tools can

greatly reduce verification turn around time, then companies would more likely adopt them

Another method is to automate the generation of testbenches with HDLs. That will greatly reduce engineers program burden; and could make language training not an issue.

## 2.6 FUNCTIONAL VERIFICATION PROBLEMS TARGETED

From our study of the functional verification domain, we find that testbench generation time has a great potential for reduction. For an IP of typical size, several engineers need to work for several months to build testbenches before starting simulations. That work includes testbench design, coding and debug.

Reusable testbench components and automatic testbench generation are key techniques to solve this problem. Chapter 3 will discuss reusable testbench in detail.

## 2.7 SUMMARY

This chapter has reviewed the important concepts in functional verification. The terminologies was defined in the beginning of this chapter. Verification approaches and important functional verification components were introduced. We also presented HVLs and discussed their advantages and limitations. At the end of this chapter we have presented our target in this project that is an environment to speed up the functional verification process, and raise the problems we need to solve.

# CHAPTER 3
# REUSABLE TESTBENCH

## 3.1 INTRODUCTION

Building testbenches is a time consuming process. It requires a good understanding of design specification, and innovative approaches to testcase development. Verification engineer need a solid understanding of the system architecture, and background in both software and hardware development. This imposes heavy requirements to testbench writers. Even, the "best" verification engineer cannot avoid human mistakes, and then testbenches themselves must be first debugged before verifying other design components. Therefore, designing, coding and debugging testbenches takes a long time, an is error prone.

Even though, some testbenches cost lots of efforts to write and debug, these testbenches typically become useless after verifying a specific DUT. New testbenches have to be developed from the very beginning to test new designs, even if they share lots of functionality with the old one. It is an extreme waste of resources. Thus, reuse of testbenches becomes a very important issue.

In this chapter, we will introduce a current approach to testbench reuse named *Aspect Partitioning for Hardware Verification Reuse* (APHVR). We will analyze its advantages and limitations. Also, we will present additional ideas to improve testbench reusability.

## 3.2 TESTBENCH REUSE

Nowadays, design reuse is very popular in complex device development. Reusing part of design code can significantly reduce development time. Since verification normally

consume 60~80% percent of effort of the total system development, reusing testbenches is essential to achieve the desired productivity.

A verification methodology that reuses testbench components is thus an obvious requirement. A key step to reuse design code is to define standard interfaces. However testbench reuse is very complex. Testbenches are used to emulate the design environment. Different systems have different functional requirements, even slight changes to the original DUT can make testbenches hard to reuse.

First we analyze testbench reusability by classifying the types of reusable testbenches.

### 3.2.1 Types of reusable Testbench components

There are several categories of reuse based on different needs. Most commonly, two classes of reuse exist: within the same project and between different projects [3]. A good example of within the same project reuse is bringing verification code from the module verification to the unit level and then the system level. Reuse between projects refers to reusing verification code from an earlier version of the design to the development of a new generation, or to a completely new system that uses standard components or has similarities to another design [3].

After studying and experimenting with verification reuse, we believe that this classification needs to be expanded. Testbenches can be divided into two kinds with respect to their reusability:

Type 1: Testbench components reusable for one specific design implementation. (Detail in Section 3.4.1.1), and

Type 2: Testbench components reusable among different design implementations (detail in Section 3.4.1.2).

### 3.2.2 Testbench Components Reuse in a Design Process

The type1, testbench components can be reused at various steps of the design process, from block level to system level. By contrast, Type2 reuse targets reuse among different design.

To perform efficient verification of complex systems, people tend to verify designs at multiple levels of integration. Thus, a testbench built for a block could expand to verify upper levels of a system.

As a block is integrated to a system, the same interface is probably used as where the block was developed. Then, some testbench components, such as stimulus generators built for a specific block could still be used for verifying upper layer of a system, as shown in Figure 3.1.For examples in Figure 3.1a), an IP is to be verified, while in Figure 3.1b), this IP is combined with other IP in a system. An example that further express this method is found in Section 3.2.2.1.

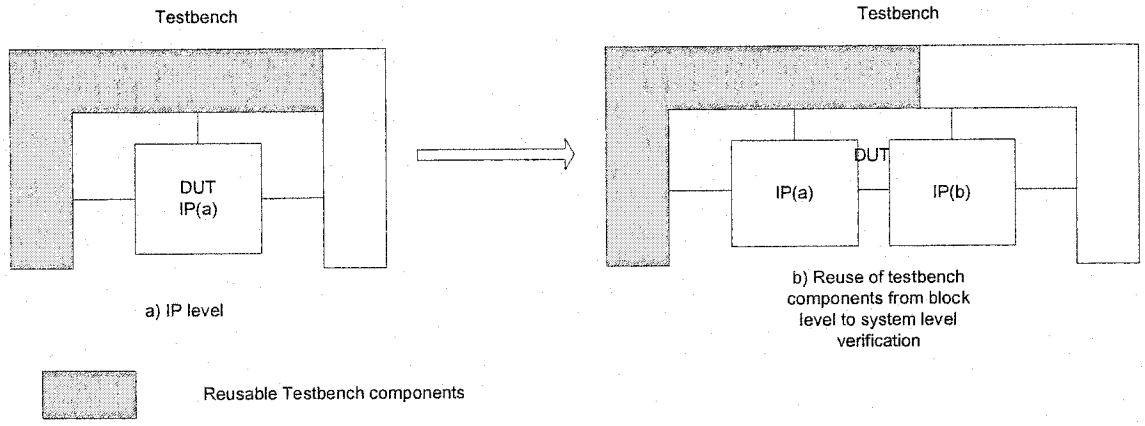



Figure 3.1: Reuse of testbench components from block level to system level verification

To reuse Type 1 testbench components, the verification environment should be well modularized. Modules should be partitioned to be less dependent from each to other. This is a classical software reuse issue. Most software reuse approaches are based on an object-oriented methodology. Feature *inheritance* of object-oriented programming promotes reuse: "You do not have to start from scratch when you write a new program. You can simply reuse existing classes that have behaviors similar to what you need in the new program "[10].

We have mentioned HVL languages (see Section 2.5) that have object-oriented feature that differ from VHDL/Verilog. Also we will present in this chapter the APHVR methodology for testbench reuse based on a HVL language, the *e* language. This method is one of the current approaches to solve Type 1 reuse issue.

In Section 3.2.3, we will introduce Type 2 reusable testbenches.

#### 3.2.2.1 Current Approach to Type 1 Testbench Reuse

The APHVR methodology is developed at the GRM (Groupe de Recherche en Micro-électronique) of École Polytechnique de Montréal. The motivation of this methodology is based on object-oriented principles to decompose testbench into modules. It uses a bottom-up approach to integrate parts of the design in a verification environment that is gradually constructed.

Aspect-oriented concepts are based on the principle of the separation of class. An aspect defines a behavior that affects several classes of a system. The object-oriented analysis organizes a system into a clear hierarchy of objects.

An aspect-oriented analysis allows a concise separation of the verification environment by defining layers in each class of the verification environment. The decomposition of the

verification environment in aspects is the core of this method. Figure 3.2 shows the partitioning categories. The base aspects block includes the definition of all these classes. Each aspect from the other categories extends these base classes. By adding features (aspect) to the base classes, other modules perform the extensions.

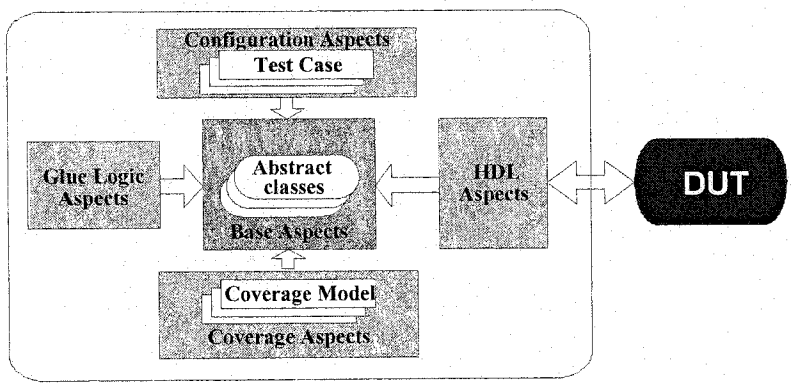



Figure 3.2: Aspect categories [8]

**APHVR example**

Figure 3.3 shows an example of a DUT that is the part of a Protocols Converter. The DUT includes five blocks: Memory Manager, Controller, Protocols Identifier, Main Multi-port Memory and Packets Input.

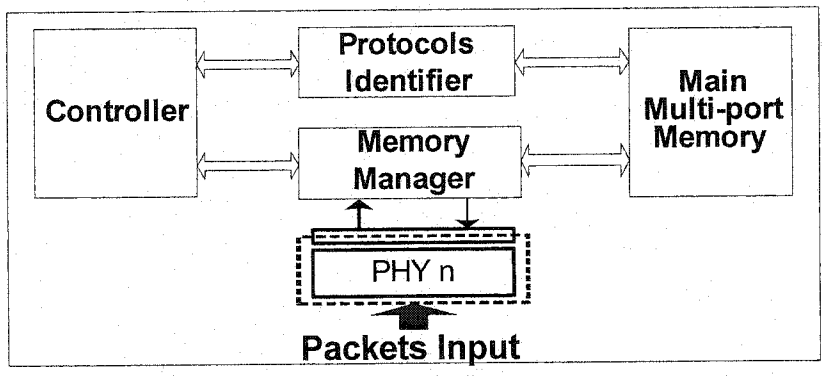



Figure 3.3: DUT: Part of the Protocols Converter [8]

Figure 3.4 represents a functional view of the verification environment for the *Memory Manager* block. It shows the main functional verification components used to stimulate the DUT and to verify its behavior.

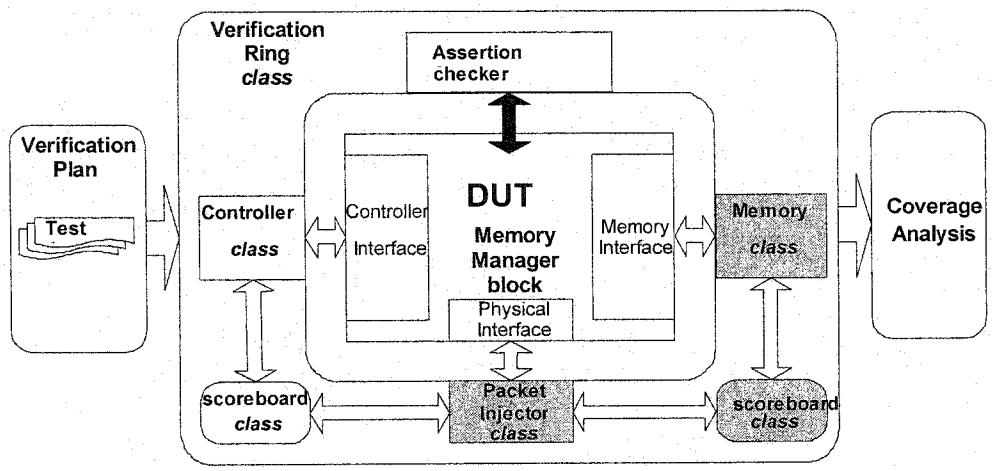



Figure 3.4: Verification Environment Functional View [8]

In Figure 3.4, we see that each functional verification component is associated with a class within the code structure. The *controller*, *memory* and *packet_injector* classes are used to emulate the *controller block*, the *memory management block* and the *packets input block* of the system. They belong to emulation type classes. The *assertion_checker* and *scoreboard* classes are part of the verification type classes.

After the *memory management block* and the *controller block* are verified separately, we combine these two blocks together. To illustrate reusability within the same project, Figure 3.5 shows a functional view of the environment with the controller block added as a second design component to be verified. In this case, a version of the *Protocols Identifier class*, which emulates *protocols identifier block*, replaces the class that emulates the *controller block* from the previous environment. Classes, such as the *packet injector class*, the *memory class*, and the *scoreboard class*, are highlighted in grey. These classes can be reused to form the new environment.

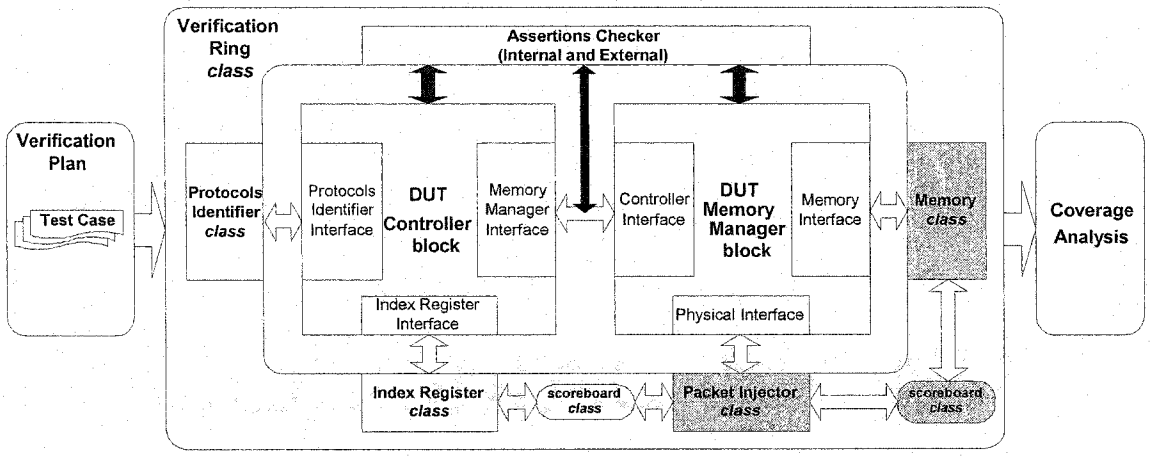



Figure 3.5: Reusability Realization [8]

**Analysis advantage and limitation of the aspect partitioning methodology**

The aspect partitioning methodology can be defined as a software reuse based on Object-Oriented Programming (OOP). It partitions testbenches at different level, and uses *inherent* features of OOP programming to extend base module to low-level. This methodology offers an efficient partitioning method and it has been shown that some aspects of verification environment possess a high reuse potential.

The target for this method is to build testbench for a specific design, and parts of testbench could be reused when this design is extended from block level up to system level, or when the same design is placed in a different system. However, the limitation is that the testbench is hard to reuse with a different design. Although a new design might be very similar to the previous one, it is still difficult to reuse testbench because of different implementation details. For example, company A develops PCI devices and builds testbenches to verify them. Later, company B decides to develop their own PCI devices and have to build testbenches again, although these testbenches perform the same functionality as the one from company A. The reason company B cannot reuse company A's PCI testbenches is that these testbenches contain specific implementation details of

company A's PCI device. It would save lots of resources for both companies if we could develop a method to reuse testbench components between those two devices.

### 3.2.3 Testbench Components Reused for Different Design Implementations

For the purpose of design reuse, many standards have been defined to facilitate integration. Buses, for example, have been defined through many standards for different usage. The reason why we need a standard is to enable integrating products from multiple venders. Every standard has a group that defines and publishes its specifications. Companies use this specification to develop their own product that can follow this well-recognised standard. *PCI Special Interest Group* (PCI-SIG) is one of the successful examples. The fundamental purpose of the PCI-SIG is to deliver a stable, straightforward and compatible standard for PCI devices. The PCI specification is an open industry standard.

Many companies have PCI products, for example, Xilinx, Compaq, IBM, etc. They all face the same verification issue. As we mentioned in the last section, these PCI devices have the same functionality, and every testbench performs almost the same function. It is not necessary that each team rebuilds their own testbench.

Testbench reuse could reduce this redundant work. Testbench components verifying certain functions for one project could somehow be reused to others. However, reusing testbench components among different implementations is not easy. First, although they all come from the same specification, different implementations may have different design features. Second, low-level implementation details differ. For instance, they often use such different signal names. The contradiction goes to that, we need testbenches general enough to suit different implementation, but on the other hand, testbenches should be specific for each implementation to verify specific design features.

To solve this contradiction, we must decompose testbenches in two parts: the non-reusable part that deals with specific implementation after features, and the reusable part for general functionality verification. As shown in Figure 3.6, since the DUTs are implemented differently, the testbench components that interact with DUT (shown as white shell) are specific to each DUT. Also, because DUTs are functionally consistent, testbench components that verify functionalities could be reused.

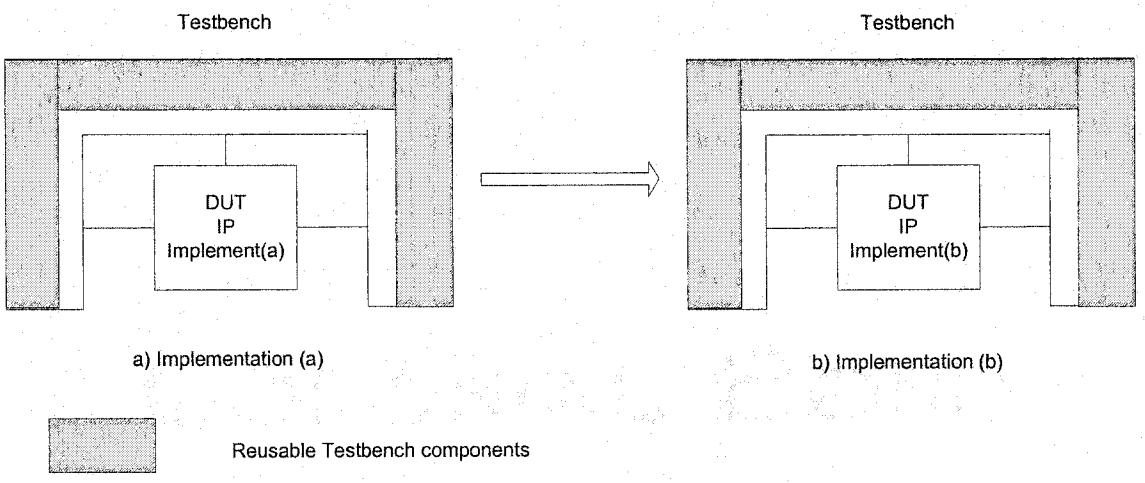



Figure 3.6: Testbench Components Reused for Different Designs

To make testbench general, we first focus our effort on specification. Because all the implements are based on the same specification, if we could capture functionality from specification, then this functionality is at high-level, the *abstract level*. This abstract level is common for all the implementations. Thus, testbenches generated from this abstract level can be reused for all implementations to verify whether the functionalities are implemented correctly according to the specification.

**Example of testbench components reused for different designs**

Let us take the PCI interface as an example. As shown in Figure 3.7, the *packet injector class* is used to emulate inputs and responses from the PCI bus, and the *PCI device class*

is used to emulate inputs and responses from PCI devices. These two classes have been divided in two parts: the functional level and the physical level.

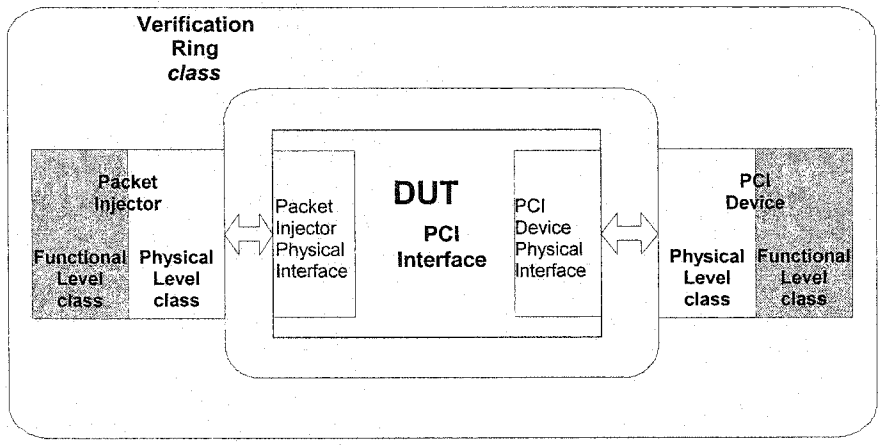



Figure 3.7: PCI device functional level verification environment

The functional level class supports general functions of the emulation environment. In this case, the functional level *packet injector class* and the *device class* emulate the device performance according to the PCI protocol. No specific low-level detail is involved. Signals and ports are defined in these functional classes as defined in the specification. These classes are reusable for any PCI interface implementation. They are shown as grey rectangles in Figure 3.7.

The physical level class is the sub class *derived* from the functional level class. Detailed low-level information on the implementation is added into this class. These non-reusable testbench components are shown as white parts in Figure 3.6 and 3.7.

In next section, we will further analyze functional abstractions.

## 3.3 FUNCTIONAL ABSTRACTION AND TESTBENCH REUSE

The more abstract is a behavior, the more general it is. This general behavior is what we need to generate reusable testbenches. How much reuse can be achieved is very dependent on the abstract level at which testbench components are built.

### The Level of Functional Abstraction

Usually, testbenches are built at the RTL level. This level is specific for implementation. Then, testbenches expressed at this level are hard to reuse because detailed implementation features are different. It is necessary to raise the abstraction level to cover more design functionality.

First, we start from specification. With a same specification, vendors could develop different implementations. We could use specification to raise those low-level implementations to a higher abstraction level. Still using PCI as an example, as Figure 3.8 shows, IBM, Xilinx or other vendors implement their PCI devices based on the PCI specification that is published by PCI-SIG. Instead of generating testbench based on the IBM implementation or Xilinx implementation as people usually do, we can design testbenches at the specification functional level. At this abstraction level, PCI behaviors are general for all PCI products. These PCI functional testbenches can be used to verify functionality of all PCI implementations expressed at the *Specification Functional Level* (SFL).

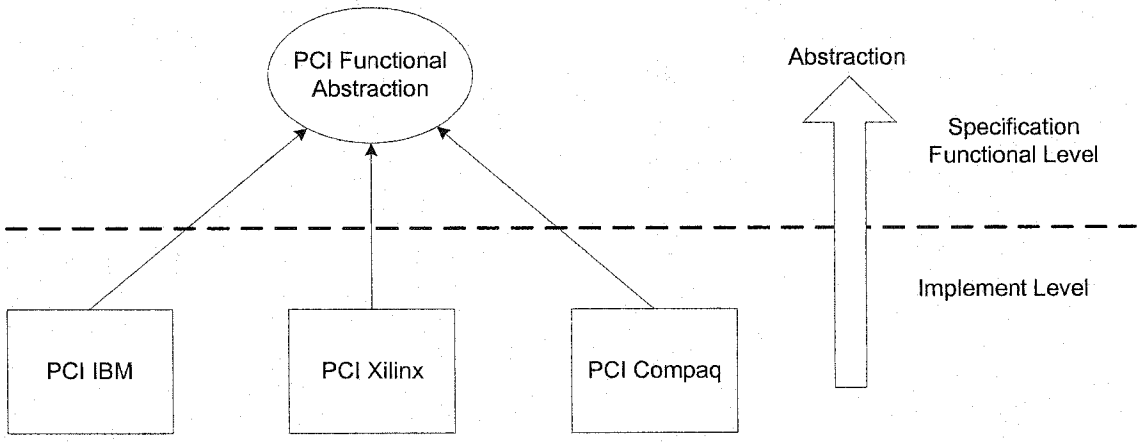



Figure 3.8: Specification functional levels

By capturing PCI specification abstraction, we could build PCI reusable testbenches to verify general functionality. This is also possible for other standards such as the *Universal Serial Bus* (USB). Then comes the question whether there is any way to reuse testbenches among PCI, USB and other standards, such as the AMBA on-chip bus. Since they are all buses, and for sure they should share some similar behaviors, it is quite possible to have one testbench to test those similar functionalities if we could capture them in a suitable abstraction.

As Figure 3.9 shows, based on the specification functional abstraction of PCI, USB, and AMBA, we could extract common features. They could be grouped together because they are all buses. Through studying enough designs in this group, common features of buses could be captured and this high-level functionality could be abstracted with functionality common to that class of modules. Thus, the abstraction is raised to a *Higher Abstraction Functional Level* (HAFL). We can repeat this process to reach a much higher-level of abstraction through studying more HAFL abstraction of different groups.

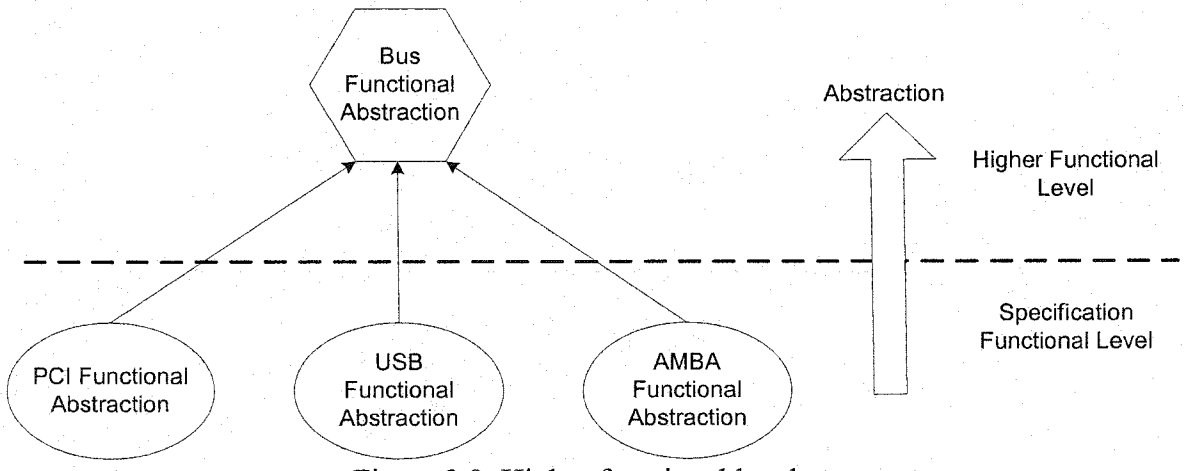



Figure 3.9: Higher functional levels

Figure 3.10 shows the whole picture of functional abstraction. At the highest level of abstraction, the more general functionality could hold. It means that it is common for many more designs. On the other hand, at the lowest abstraction level, design specific features can be verified. This abstraction level has more detailed design information.

According to this functional abstraction tree, we could build testbenches at different functional level aiming to verify different levels of design functionality.

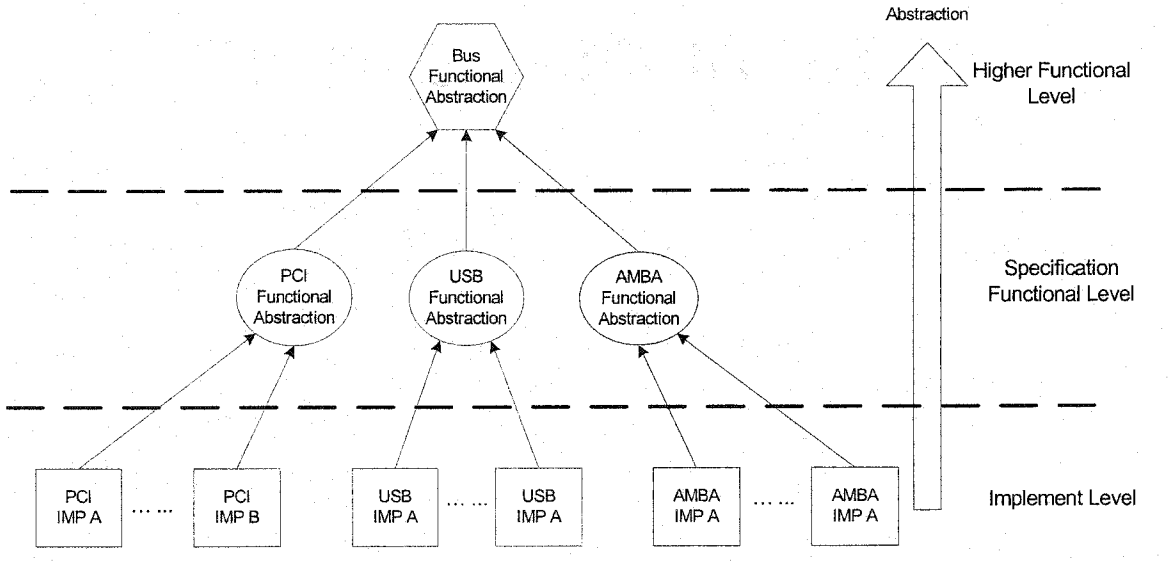



Figure 3.10: Whole picture of functional abstraction

Low-level testbench modules could be easily *derived* from high-level abstraction with object-oriented inheritance features. That could decrease abstraction level to the lowest level, implementation level, to verify specific implementation features. When we want testbench components to be reused by more designs, we should build them at the higher abstraction level. When we want to verify implementation specific features, we decrease the testbench abstraction level to the lowest level.

We first focus our effort on the SFL abstraction in this project. In Chapter 5, we will introduce a methodology to generate reusable testbenches based on SFL abstraction. Before that, in Chapter 4, a method to capture SFL abstraction will be presented.

## 3.4 SUMMARY

We discussed the importance and the necessity of verification reuse in this chapter, and we presented our research goals.

From the above analysis, we can conclude that reusable testbenches can be classified into two types. Type 1 reuse shares components for different phases of a design, while type 2 reuse tries reusing components with several designs.

We introduce a current approach, the APHVR methodology, to accomplish type 1 reuse. This chapter also presents and analyzes functional abstraction methods, which is our contribution in this project, in order to perform the second type of testbench reuse between different designs.

# CHAPTER 4
# REUSABLE TESTBENCH GENERATION METHODOLOGY

## 4.1 INTRODUCTION

In the last chapter, we have mentioned that the first abstraction level is the functional level specification. It abstracts the implementation. Because this functional abstraction is based on the design specification, we will start by the capture of functional specifications. More specifically the system is to be described as a functional specification.

There are several languages available to realize this task, such as SDL [29], UML or SystemC. In this work, we have selected SDL. In this chapter, we will motivate this choice, first by introducing the features of SDL and secondly by comparing SDL to UML and SystemC.

This chapter introduces our testbench-building methodology in order to automate the generation of testbenches. The methodology supports both types of reuse. We will focus on type 2 (section 3.4.1.2), more precisely the testbench reuse among different designs. Also, as mentioned previously, the level of abstraction of our methodology (Section 3.5.1) is the first abstraction level – the specification functional level.

## 4.2 SYSTEM FUNCTIONALITY CAPTURE BASED ON SDL

The SDL is an object-oriented, formal language defined by The International Telecommunications Union – Telecommunications Standardization Sector (ITU-T) as recommendation Z.100 [30].

SDL possesses a rich grammar that describes its behavior. Also this grammar is unambiguous. Therefore, it is possible to build tools for the simulation of SDL systems

and for the validation of formal characteristics, such as deadlock avoidance. This means that many errors can be detected at a very early stage. This feature makes SDL an executable specification language.

The SDL language is intended for the formal specification of complex, event-driven, real-time, and interactive applications. They involve many concurrent activities that communicate among each other by using discrete signals. SDL has been designed for the specification and description of system behaviors, including the interaction of the system and its environment. It is also intended for the description of the internal structure of a system, so that systems can be developed step by step.

Figure 4.1 shows four main hierarchical levels of an SDL system. In a SDL specification, a *system* consists of numbers of blocks connected by channels. A *block* is an enclosure for the further structuring of the system. Channels connect the blocks with each other and with the environment of the system in a one-way or two-way mode. A block consists of *processes* connected by signal routes. Each *process* is an extended finite state machine (EFSM). These machines (or processes) run in parallel. They are independent of each other and communicate with discrete messages, called *signals*. A process can also send signals to and receive signals from the system environment. The behavior of a state machine is characterized by a set of transitions. A transition to another state or the same state occurs whenever a stimulus (or input) is consumed. When a process is in a state, it accepts stimuli from its input port. These stimuli can be signals received by input ports or timers. When a process enters into a new state, it means that a transition terminates. Based on the value associated with a variable, EFSM enables decisions to be made in transitions. Then, the state, which follows when a specific input is consumed, is only determined by the existing state and input. A transition may contain the following actions:

- *Output*: to send signals.

- *Task*: to change the value of variables. Local variables for each machine may contain details about the history of the machine.
- *Create*: to create process instances.
- *Decision*: to split into several sequences of actions.
- *Call*: to activate procedures. A procedure is a parameterized part of a process with its own scope.
- *Set, reset*: to manipulate timers.

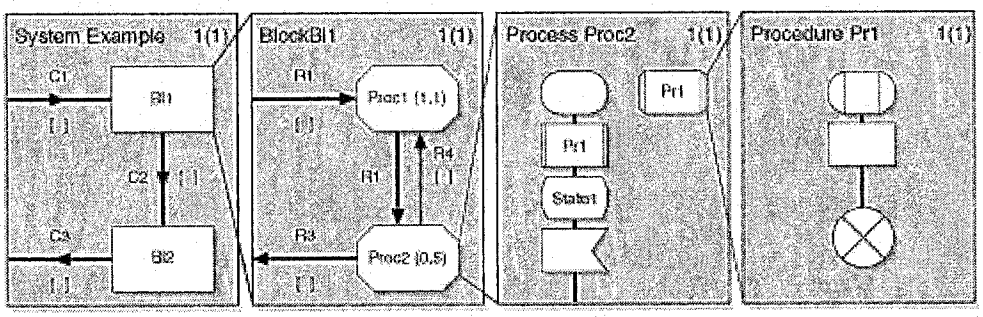



Figure 4.1: Four main hierarchical levels of a SDL system [30]

The SDL language supports two equivalent notations: the graphical notation (SDL-GR) and the textual notation (SDL-PR). The graphical notation (SDL-GR) is a standardized graphical representation of the system. SDL elements such as system, block, process, signal etc. are drawn by using standardized graphical symbols. The textual notation (SDL-PR) is a textual representation of the SDL system, or in other words, it is a SDL "source code" [31].

### 4.2.1 Choice of SDL as a Specification Language

We have discussed the system functional abstraction in the previous chapter. In our methodology, we need a language to capture this abstraction, and this language should provides the following services:

First, it should allow to capture system functionality. This language should describe the system as a specification. Then, system abstraction could be captured at the functional

specification level (see Section 3.5.1). Such a system specification should be executable in order to be validated before being used to generate testbenches.

Secondly, it should work as a standard input for automatic generation tools. Indeed, one of our objectives is to develop a tool to automate testbench generation based on our reusable testbench development methodology. Information captured from the functional level system specification is fed into the tool in a standard format to allow building testbenches.

Thirdly, to represent system behaviors with EFSMs, which is very important for testbench generation, and synthesis of coverage model.

UML and SystemC could also be used to capture system functionality. They are similar to SDL in some features.

We will compare these three languages in the following sections.

#### 4.2.1.1 SDL and UML

SDL is increasingly used with UML: UML is well adapted to the analysis of high-level design phases of a project, while SDL-92, formal and powerful to describe actions, is better for subsequent phases and detailed design. Most SDL-92 tools provide an automatic conversion of UML diagrams into SDL.

SDL textual language is an executable language. UML, however, cannot be executed. SDL possesses simulators and test generators to verify SDL descriptions. These features are useful to make sure that SDL models are equivalent to the specification.

Despite the difference between SDL and UML, they are still very similar. In the future, there are plans to merge these two languages as illustrated in Figure 4.2. SDL-2000

integrates UML features such as Class Diagrams and State Charts Diagrams: when SDL-2000 supporting tools are available, analysis and design will be done in one tool, providing both UML and SDL views of the same model.

UML 2.0 promises to merge UML and SDL into one powerful language that is suitable for the development of real-time/embedded systems, as well as for software development in general [35].

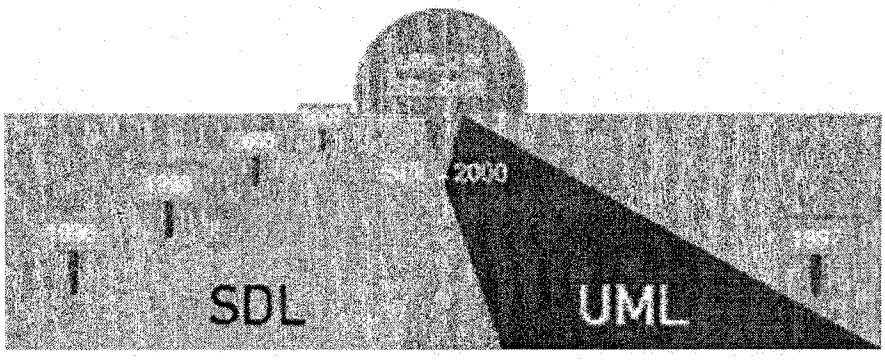



Figure 4.2: The Future of SDL and UML [35]

#### 4.2.1.2 SDL and SystemC

SystemC is a standard design and verification language built in C++ that spans from concept to implementation in hardware and software. Engineers design and verify using SystemC and the ANSI C++ standards.

SystemC provides innovative mechanisms for C++ based on system-level descriptions. It enables, promotes and accelerates system-level IP model exchange and co-design using a common C++ modeling platform [38]. It is one of the best choices to develop an executable model of a whole system architecture to gain confidence in decisions that have been made [39]. In the following, some important features of SystemC are presented:

- Modules: This is a hierarchical entity that can contains other modules or processes. Those Module and processes can have a functional interface, which allows the entity to hide implementation details.
- Processes: Processes are used to describe functionality. SystemC provides three different process abstractions: asynchronous blocks, synchronous and asynchronous processes.
- Signals: SystemC supports resolved and unresolved signals.
- A rich set of signal types: To support modeling at different levels of abstractions, from the functional to the RTL.

The above features of SystemC are very useful to describe systems as executable specifications. However, SystemC does not support EFSM. As mentioned above, as a feature of SDL, EFSM is extremely convenient for verification engineers to develop functional coverage models. This feature of SDL makes it better suited for the task, because in this project, we focus on the relation between specifications and testbench generation.

### 4.2.2 Conclusion of the Comparison

SDL, UML and SystemC are not equally suitable for the capture of system functionality, however all of them could be used. Their specific features make them better adapted for different requirements.

In our methodology, the differences among those languages make SDL a good candidate. It provides an executable system specification that supports EFSMs. As a second reason, this work is part of a larger project supported by an industrial partner, PMC-Sierra, and SDL is used in PMC-Sierra's design methodology. Consequently, PMC-Sierra encouraged us exploring SDL to capture executable specifications in this project.

SDL is our first choice in this project. The core of this tool defines what information is needed but does not specify which language retrieves this information. That makes the tool open to various methods of capturing abstraction as long as the tool recognizes the information format.

### 4.2.3 SDL System Abstraction and Testbench Building

The goal of capturing system abstraction is to build testbenches. In this section we will present how the information described in SDL is used to build testbenches. We use a concrete example to illustrate the relation between SDL system description and detailed testbench building.

#### 4.2.3.1 A Concrete Example

We will use a Quad ATM user-to-network interface and forwarding node (SQUAT) as an example to develop our methodology [34]. Figure 4.3 shows an overview of the design. UNI ATM cells received on four input interfaces are reformatted as NNI cells. They are then routed to the appropriate output interfaces. The host processor in an internal register file maintains the rewriting and forwarding information. The total number of ATM cells dropped due to output interface congestion is continuously reported in an internal status register.

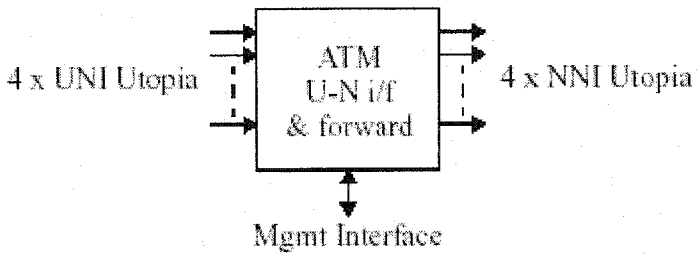



Figure 4.3: ATM switch overview diagram [34]

Figure 4.4 shows how the information flows within the design.

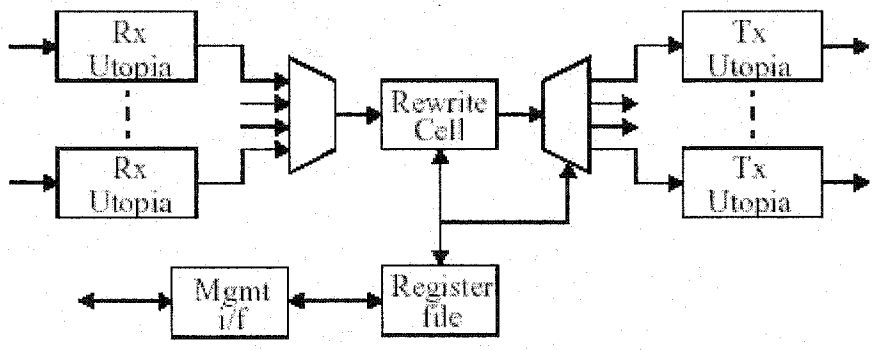



Figure 4.4: ATM switch structure diagram [34]

The design translate cells from the UNI format, as shown in Figure 4.5(a), to the NNI format, as shown in Figure 4.5(b). The only fields used by the design are the VPI and HEC fields in the incoming cell. The only fields modified by the design are the GFC and VPI fields in the incoming cells, which are replaced by the VPI value of the outgoing cell, and the HEC field of the outgoing cell that is recomputed.

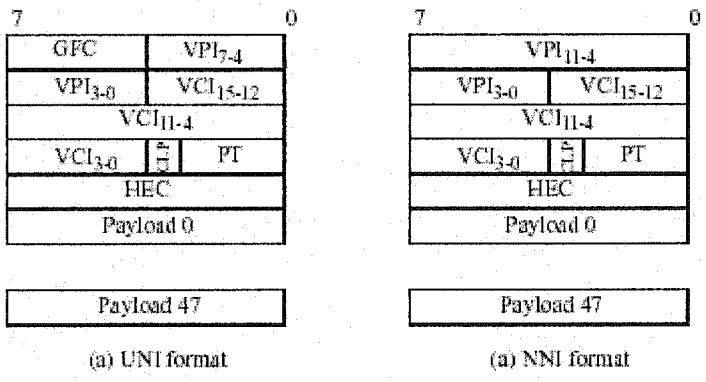
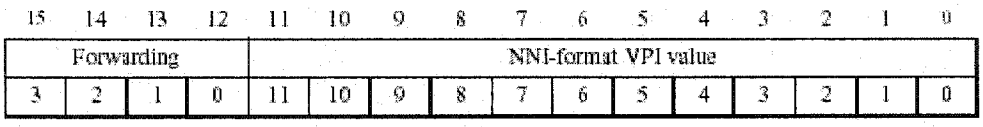



Figure 4.5: UNI and NNI cell formats [34]

The register file contains 256 16-bit values with the rewriting and forwarding information for each UNI VPI value, from 0x00 to 0xFF. The 16-bit rewriting and forwarding configuration for a given VPI has the following format:




The most-significant nibble is the forward mask. It is used to forward cells with a UNI VPI value corresponding to that configuration register to the specified output ports (after rewriting). The remaining bits in the configuration register contain the NNI VPI value corresponding to the incoming cell's UNI VPI value. It replaces the GFC and VPI fields with the incoming cell.

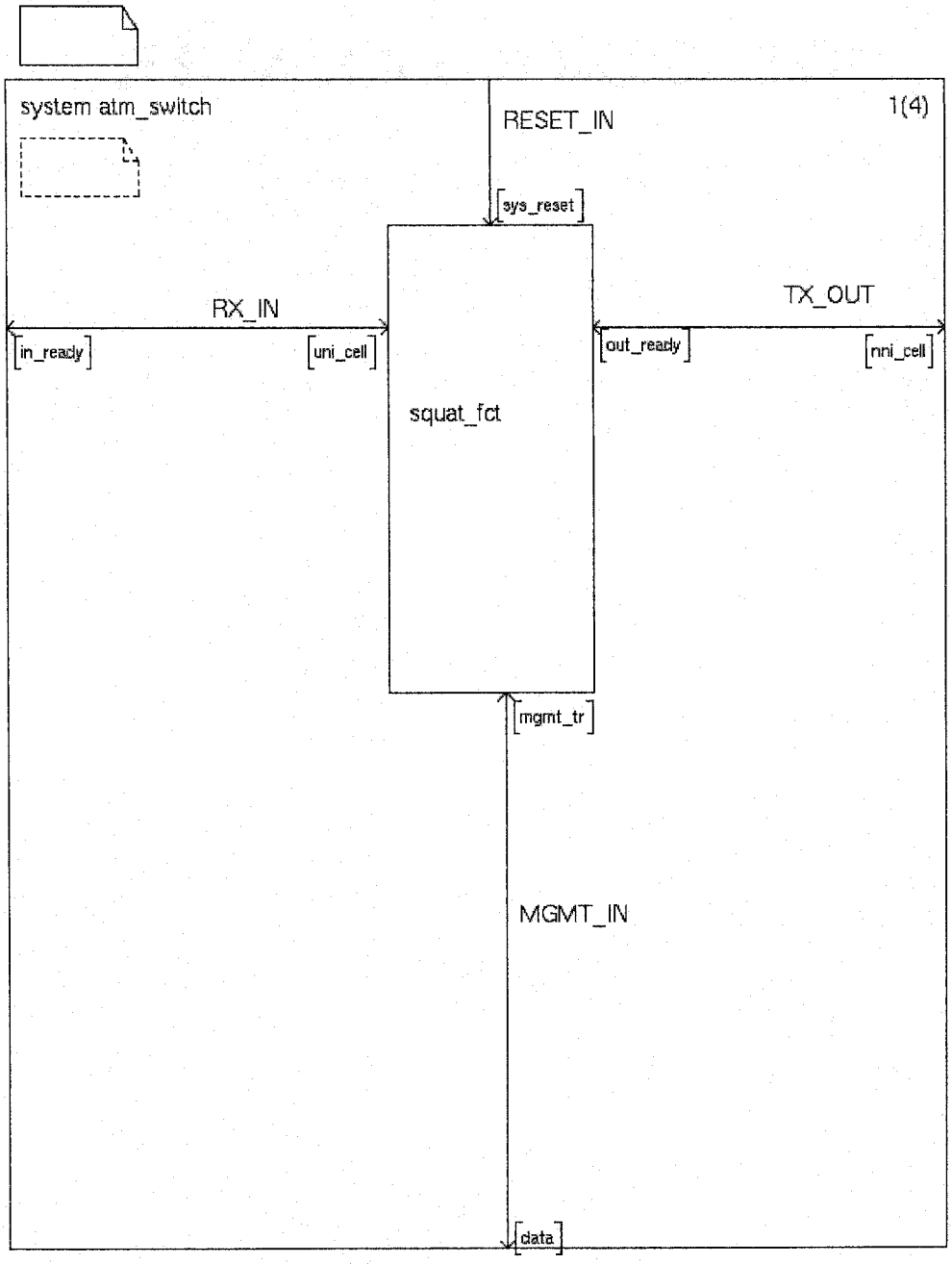



Figure 4.6: SDL system level descriptions

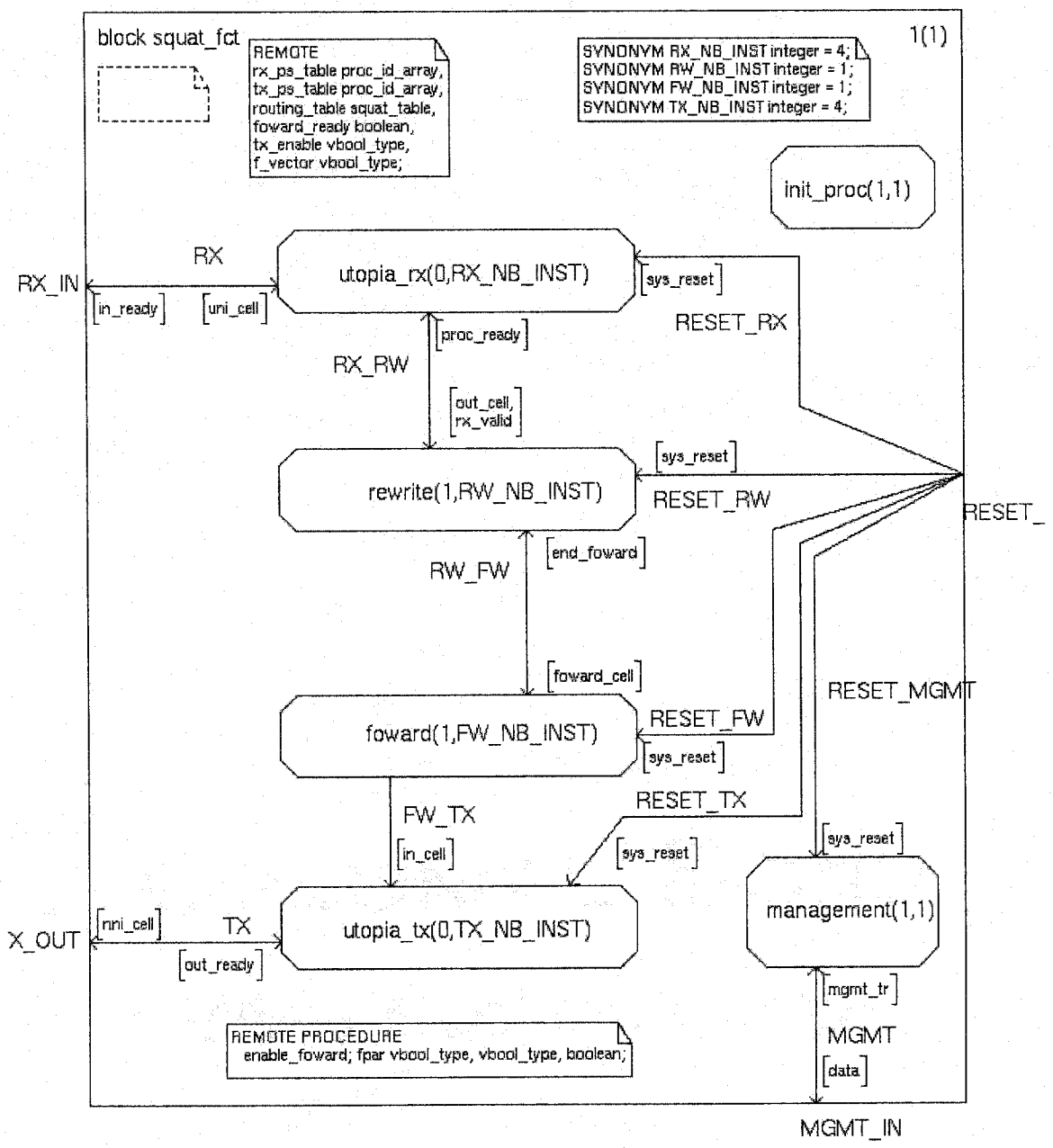



Figure 4.7: SDL block level descriptions

#### 4.2.3.2 An Executable SDL Description

We could describe our example with a SDL description. Figure 4.6 shows the top level of the SDL description for our example. Figure 4.7 shows a typical SDL description of a block. It contains more detailed information than a system level description.

The goal of capturing system abstraction is to build testbenches. In this section we will present how the information described by SDL is used to build testbenches. In SDL Graphic language the system description is called a *system diagram*.

A SDL diagram usually contains the following elements:

- *System name* (atm_switch in Figure 4.6);
- *Signal descriptions* provide the types of signals interchanged between the blocks of the system or between the blocks and the environment (see uni_cell, nni_cell, etc in Figure 4.7);
- *Channel descriptions* provide the connections between the blocks of the system and to the environment (see RX_IN, TX_OUT, etc. in Figure 4.6);
- *Data type descriptions* provide the user defined data types visible in the whole system and its environment (in Figure 4.7);
- *Block descriptions* provide the blocks into which the system is partitioned (see squat_fct in Figure 4.6).

The main function of a testbench is to emulate the environment of a module system. From the SDL system diagram (Figure 4.6), we could clearly see the inputs, outputs between modules and to the environment. We can decide that a testbench needs:

- An injector to send the *RX_IN* signal into the DUT randomly.
- A monitor to handle output *TX_OUT*.
- A module to emulate *management* input MGMT_IN.
- And reset signals.

According to this information, we can define the structure of the testbenches as shown in Figure 4.8. In this testbench, three emulation modules need to be generated: packet injector, monitor and management. These three emulation modules provide inputs and outputs (Shown as SDL diagram Figure 4.6). Reset signals are inputs only.

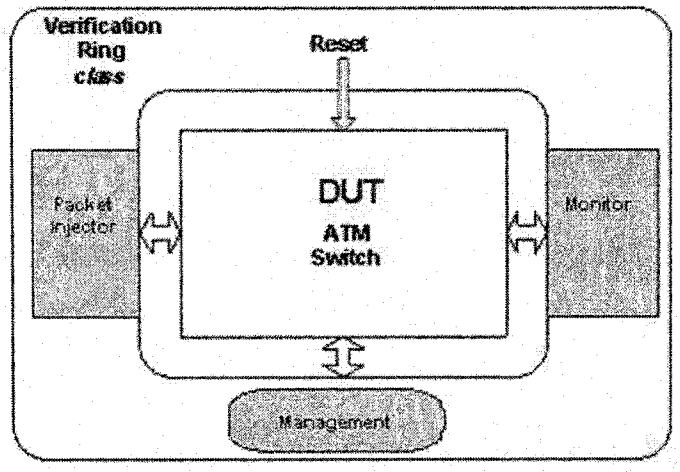



Figure 4.8: Verification environment of DUT ATM switch

The data type of injected stimulus could also be generated according to SDL *Data type descriptions*. These high-level signals should be mapped into specific signals used in the implementation during the DUT simulation. Further details are provided in the following sections.

Thus, from SDL diagrams, we could get all the information necessary to build testbenches for the system.

## 4.3 A METHODOLOGY FOR BUILDING REUSABLE TESTBENCHES

### 4.3.1 Overview of the Methodology

Figure 4.9 shows the whole picture of the methodology. First, we need to capture the system functional abstraction using SDL. After the specification has been specified and verified, the development team could be divided into two groups: the design team that will implement the specification into RTL level and the verification team that will build the testbench. Performing the work in parallel could greatly reduce the development time compared to the situation where the developers would wait to write testbenched until the RTL description has been completed. Also, testbenches developed at this stage are not based on implementation details, since the information used to produce them is a SDL high-level description. Thus, the testbench components are more likely reusable to verify other implementation of the same functionality.

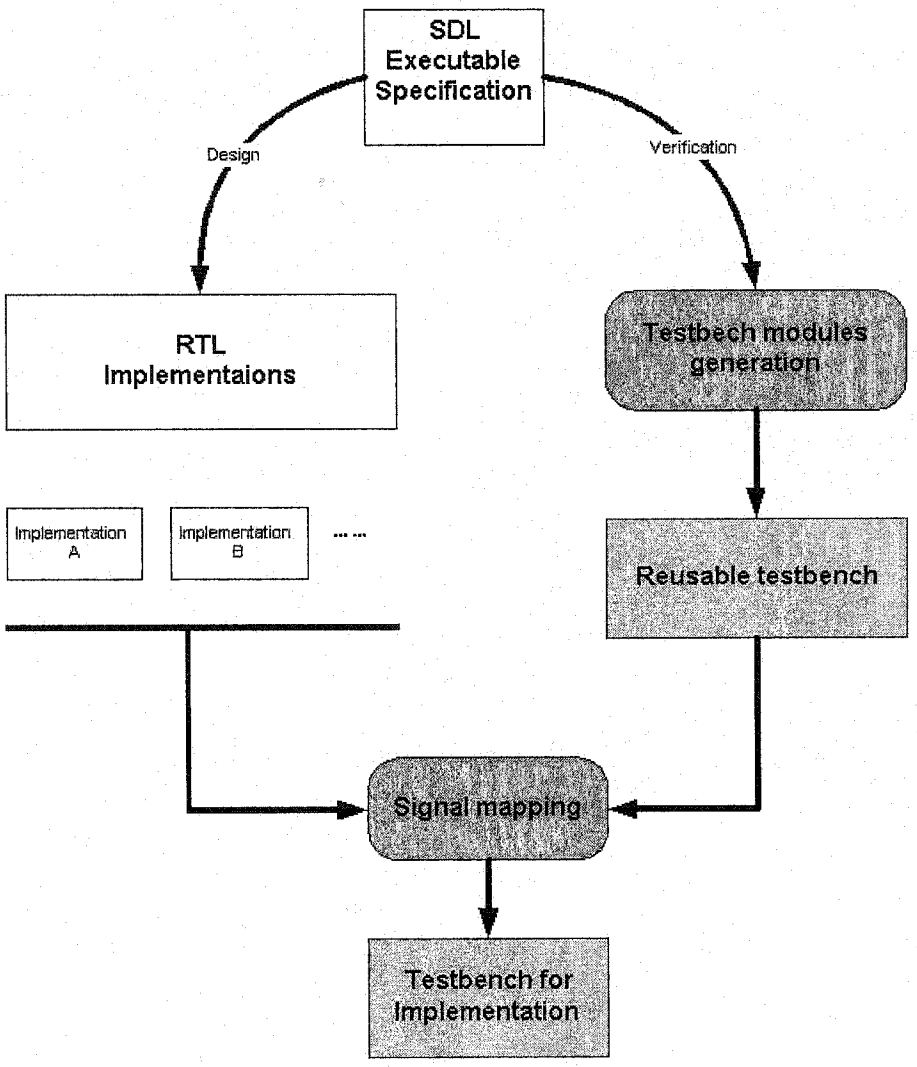



Figure 4.9: Overview of verification methodology

During testbench components generation, the testbench need to be well partitioned into modules following the APHVR methodology (Section 3.3.1). That provides type 1 testbench reusability, and leaves room for future expension with features of OOP.

Before using the testbench to simulate the DUT, we need to map this testbench into a specific design implementation at lower level through signal mapping (details of signal mapping will be introduced in the following section).

Then, after signal mapping, we have a testbench ready to begin verification. In the rest of the chapter, we present details of each step of the methodology with the same example, the ATM switch presented in Section 4.2.3.1.

### 4.3.2 Functionality Capture

In Section 4.2.3, we introduced the relationship between SDL specification and testbench generation. Because a testbench generation tool will be developed to implement this methodology, instead of being understood by human, specification should also be recognized by this tool. This specification must be formatted as a standard input.

#### Retrieve SDL Information form SDL Specification

The first step to retrieve information from SDL specification is to convert the SDL graphic presentation to textual notation. We use an existing tool to perform this step, namely Tau form Telelogic [35].

#### Telelogic Tau SDL Suite

The Tau SDL Suite has been developed by Telelogic to simplify and speed up SDL development work. The SDL Editor is a modern, graphical drag-and-drop editor, as

shown in Figure 4.10. Its SDL graphic notation can be automatically converted to textual notation.

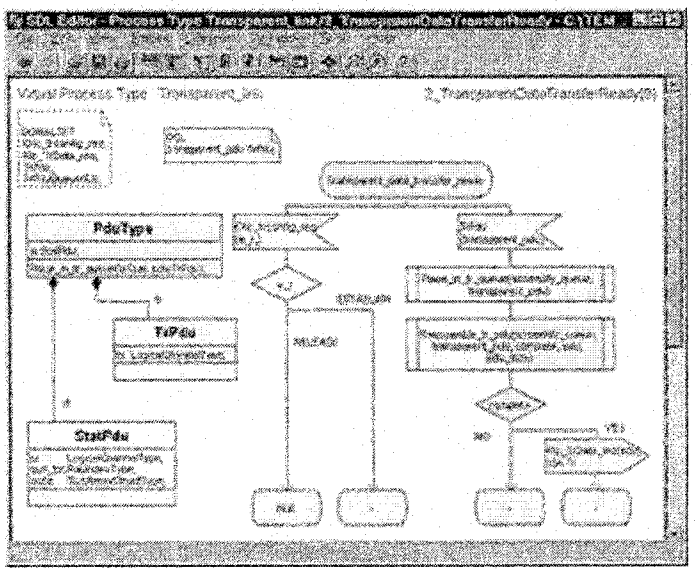


Figure 4.10: Tau SDL graphic editor [35]

Figure 4.11 shows the SDL textual description for the SDL diagram of Figure 4.6. In SDL textual notation, the keyword *system* defines a system; between *channel* and *end channel* that define the source and destination of a channel with certain types; *block* defines a block; and *signal* is used to define signals. From these keywords, we could retrieve the same information as we did in Section 4.2.3.2 to define verification environment. The difference is that with textual language, we could develop a tool to automate information retrieved. We name this tool *parser*, to scan and parse SDL language in order to retrieve the required information.

```
system atm_switch;                          // System name
channel RESET_IN nodelay                    // define channel
  from env to squat_fct with sys_reset;     // source/destination/type of channel
endchannel RESET_IN;                        // end of channel definition
channel RX_IN nodelay
  from squat_fct to env with in_ready;
  from env to squat_fct with uni_cell;
endchannel RX_IN;
```

```
channel TX_OUT nodelay
  from squat_fct to env with nni_cell;
  from env to squat_fct with out_ready;
endchannel TX_OUT;
channel MGMT_IN nodelay
  from squat_fct to env with data;
  from env to squat_fct with mgmt_tr;
endchannel MGMT_IN;
block squat_fct referenced;                    // define block
signal                                         // define signal
  uni_cell(atm_cell_uni),
  nni_cell(atm_cell_nni),
  in_ready(integer),
  rx_valid(integer),
  out_ready,
  proc_ready,
  out_cell(atm_cell_uni),
  end_foward,
  foward_cell(atm_cell_nni, vbool_type),
  in_cell(atm_cell_nni),
  mgmt_tr(mgmt_trans),
  data(squat_line),
  sys_reset;
```

Figure 4.11: SDL textual notation

As Shown in Figure 4.12, the input of the *parser* are SDL textual files, and the output is a data-structure that contains all the information necessary to generate testbenches. Within the *parser*, the required data is predefined, such as channel name, type and the source/destination of this channel.

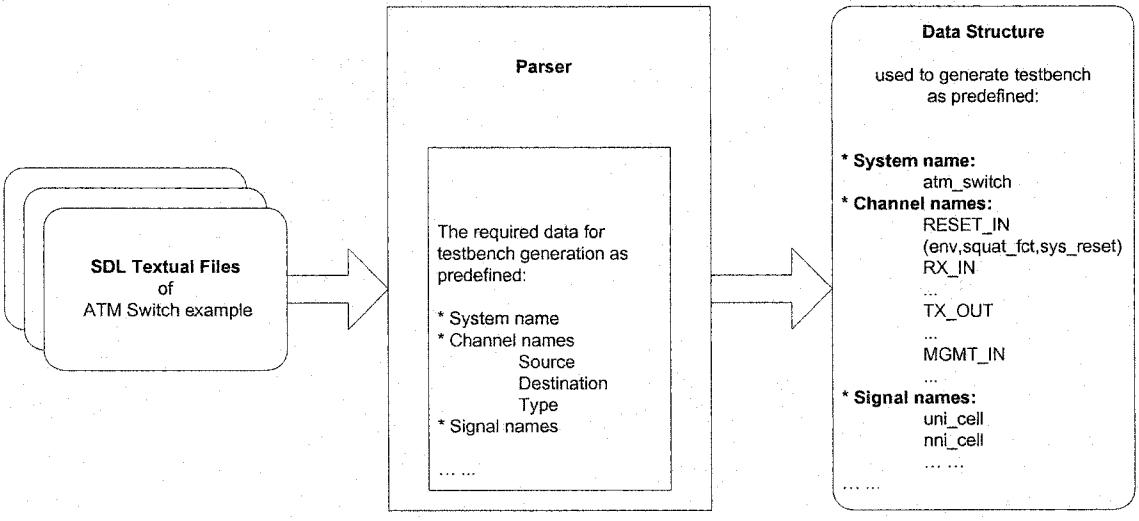



Figure 4.12: Structure of the parsing tool

### 4.3.3 Generation of Testbenches at Functional Level

Before generating testbenches, we should know what information is needed, where to get this information, and how to use it.

**Type of information required**

To build testbench components, at least the following points are required:

- Number of inputs and outputs
- Number of inputs of the same type

Extraction of this necessary information should be implemented in the *parser*.

**Location of the information**

Figure 4.7 shows a block level SDL description. It contains more detailed information than system level description. From this, the following questions could be easily answered.

- Number of inputs and outputs
  - One input channel RX
  - One input channel MGMT_IN
  - One reset channel RESET
  - One output channel X_OUT
- Number of inputs of the same type
  - "SYNONYM RX_NB_INST integer = 4;" this SDL statement defines that channel RX has four instances
  - "SYNONYM TX_NB_INST integer = 4;" this SDL statement defines that channel TX has four instances

The *Parser* retrieves this information and generates data structure to build testbench.

**How to use the information to build the testbench**

We use the APHVR methodology to generate testbenches in the *e* language. As shown in Section 4.2.3.2, from a SDL specification we can define a verification environment (Figure 4.8). For instance, we will present one of the required emulation modules, *injector*, to show how testbenches can be generated with *e* language.

```
  --File: atm_switch_injector.e
    module: atm_switch injector
    aspect: base
… …
<'
unit atm_switch_injector{ -- define unit
-- Static
   cells    : list of atm_cell_uni;     -- define data
   rx_data_in     : string;                  -- define input port

… …
   send_transaction(t: atm_cell_uni)@sys.driving_clk is empty;
   -- send stimuli into DUT
… …
       all of {
              {
                 for each (b) in t do { --send data as input of DUT
                    '(rx_data_in)' = b;
                 wait cycle;

              };
            };

… …
   do_reset() is empty;
   -- do reset
};
'>
```

Figure 4.13: Source code of testbench with *e* language

Figure 4.13 is the source code segment of a testbench for the injector module. The bold word is what we retrieved with the *parser*. In the *e* file, an injector *unit* combined with a concrete system named, *atm_switch*, is generated. The function of this *unit* is to inject stimuli into the DUT with the function send_transactionAgain, a stimuli type, *atm_cell_uni*, which is retrieved by the parser from the SDL specification, replaces parameter of this function. The *for* statement, "`'(rx_data_in)' = b;`" sends streams of data into the input port, *rx_in*, of DUT. Port name *rx_in* is defined in SDL specification as input *channel*. Note that the port name, *rx_in* is only defined in the specification. In a specific implementation, it might be of different name. Mapping functional level signal to

a specific implementation needs to be performed before a testbench can be used to simulate a DUT.

### 4.3.4 Mapping functional testbenches at low level

After the design has been implemented into RTL, we need to extend our functional testbenches to low level by deriving base modules with the actual signals names. As shown in Figure 4.14, a low level module, *unit atm_switch_injector*, is derived with the *e* language *extend* keyword. This replaces the generic name from the SDL specification by the actual port name.

```
--File: atm_switch_HDL_low.e
 module: atm_switch HDL_low
 aspect: HDL
... ...
<'
extend atm_switch_injector{
  keep soft rx_clk_out == "clock name";
  keep soft rx_data_in == "input port name";
  ... ...
};
'>
... ...
```

Figure 4.14: Mapping signal name to a specific design

## 4.4 FEATURES OF THE METHODOLOGY

In Section 4.3, we have introduced a methodology to build reusable testbench. Through verification reuse, redundant testbench components generation can be reduced. Also, our methodology supports features that are different from traditional verification method: specification for verification, executable specification, and concurrent design and testbench generation.

### 4.4.1 Specification for Verification

As verification plays more and more an important role during the system development, while spending more resources than the design does, we need to consider this fact. When developing specification, we should also pay attention to verification instead of only considering design and implementation. Indeed, traditionally, the verification aspect is not considered during the specification development. However, it is necessary to consider verification at the beginning stage of the writing specification. In our approach, testbench generation is based on the specification and this specification works as an input of the *Reusable Testbench Generation Tool set* (RTGT).

First, to reduce the human interaction, the RTGT tool needs to write specification with a certain format that could plays the role of a standard input. This requires the specification written using a formal language, with a strict syntax that can be easily analyzed by the tool RTGT.

Secondly, during its written, the specification must include information to improve the automation of the tool features. For example, to automate the generation of the assertion checker, relative information with a certain format should be added into the SDL specification. Then, the parser can easily retrieve the necessary data. This requires designers aware of the verification. Also, the embedding of information for verification will consume more time, however, the benefit is very attractive because in this way, the tool could obtain enough information from specifications, to achieve automatically most of the testbench generation.

### 4.4.2 Executable Specification

One of the advantages of writing specification with SDL or SystemC is that specification could be verified or validated itself. The idea of executable specification has been raised twenty years ago, and it is still today an important research area.

If you consider SDL as an example, the Telelogic Tau SDL Suite offers a rich set of tools for simulation, both for verification and for validation. The verification is performed not only through static analysis, but also through an automatic exhaustive exploration of the SDL specification. This is done by testing all possible execution paths of the SDL system, during which a number of rules are checked and violations are reported, e.g. deadlocks, loops and exceeded queue length [35].

It is very important to verify the specification because all our implementation and verification are based on them. We must make sure that this specification is validated before moving to the next step.

### 4.4.3 Concurrent Design and Testbench Generation

Traditionally, designers first implement specification at a Verilog or HDL RTL level, and then write testbenches after. Thus, verification engineers should wait till design has been implemented. Obviously, in order to reduce the development time, a concurrent approach, where design and testbench generation occur at same time, is required.

The pipeline of our approach, that supports concurrency, is shown in Figure 4.15:

Step 1: The design team implements the specification in HDL. The first step is to define data (signals, ports, etc). At the same time, verification team generates functional testbench components.

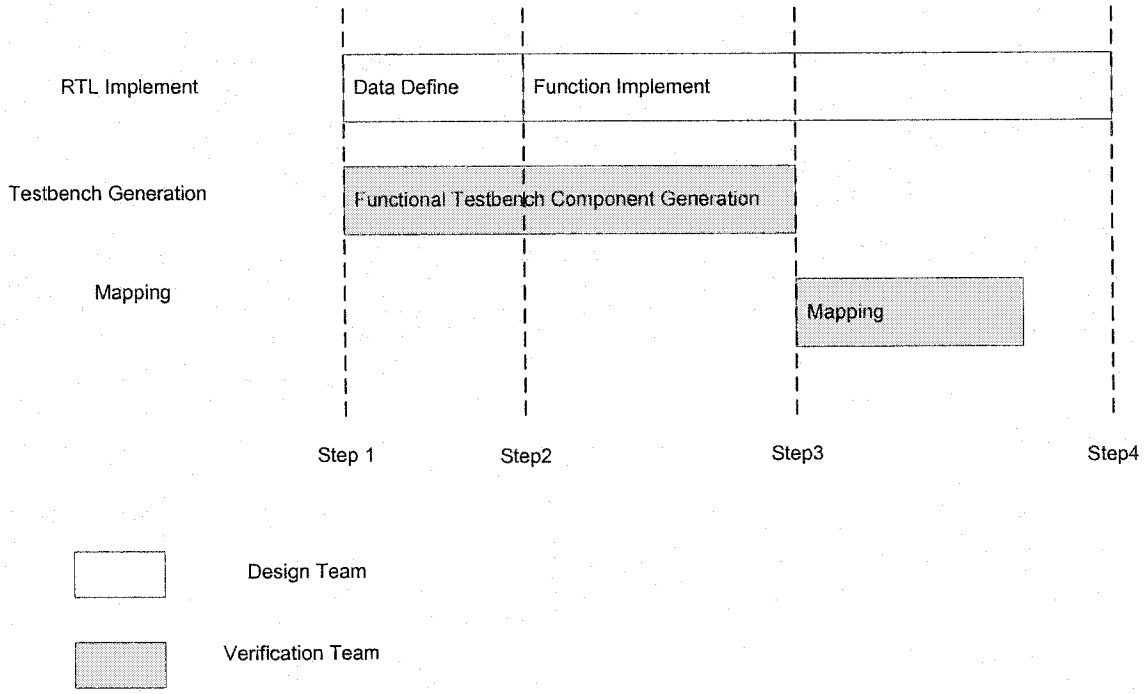



Figure 4.15: The pipeline of design and testbench generation

Step 2: The design team has done the data definition and continues to implement functions. At this stage all the signals have been defined and are ready to be mapped into the testbench by the verification team. However, this mapping process would not start till high-level testbench modules have been completed.

Step 3: The design team is still implementing HDL function. The high level testbench has been done. The verification team uses data defined by the design team, to map the high level testbench at a lower level.

Step 4: The design and the testbench are completed. Simulation can start to perform functional verification.

## 4.5 SUMMARY

In this chapter, we have presented SDL, UML and SystemC languages that could be used to capture system abstraction. A comparison has been made among these three languages to show their own specific design features. According to our requirements, and the specific features of language, we have decided to use SDL as system abstract language.

Also, an overview of SDL language, and an example showing the potential of SDL as testbench generator, has been presented.

We have also presented a methodology of testbench reuse, and described a concrete example. The main steps of the methodology are: system functional abstraction capture, information retrieving, information processing to build testbenches, and the mapping of high-level testbench with low-level data.

This methodology supports two types of testbench reuse. 1) Reuse among different design implementations, the main contribution of this project, and 2) reuse method for one specific design implementation. The reusability exists at two levels: before the mapping, where testbench components could be used by any design implementations, and after the mapping, where testbench components can be used into a specific implementation when DUT is raised to system level. With this reusable testbench methodology, and its automated tool, the testbench generation time could be greatly reduced.

Finally, we have presented features of this new methodology, which are: specification for verification, executable specification, concurrent design and testbench generation.

In the next chapter, we will present *rule* technique to implement this methodology into an automated generation tool, named RTGT.

# CHAPTER 5
# IMPLEMENTATION OF THE METHODOLOGY

## 5.1 INTRODUCTION

In this chapter, we will introduce a way to develop tool RTGT. *Rule* technique is used in this tool's development. What are rules, why use rules, and how the rules work, will be discussed in detail in the following sections. Also, we will present the architecture of this tool and its implementation in C++.

Finally, we will focus on some questions related to the performance of the tool.

## 5.2 FEATURES OF THE TOOL

The next three points present features allowing the speedup of the verification process supported by our tool.

### 1) Testbench reuse

The testbenches generated allow reuse from IP level to system level and reuse among different designs (see Chapter 4).

### 2) Top-down partition and bottom-up verification

Top down design is a methodology used frequently by the industry. The design flow can be divided into the following steps:

- Architectural exploration

For an implementation, system engineers first analysis specification and then partition system into several blocks. Afterwards, decision should be made regarding which blocks should be mapped on commercial IPs and which blocks need to be developed by the team.

- Refinement

  After obtained high-level block from the previous step, engineers could refine partitioned blocks and then define functions (processes).

- Definition of functions

  Engineers code these functions (processes) with certain languages.

After the coding has been done, all the work should be integrated together. Usually, at each level of integration several engineers are involved in the code writing. Each coding should be verified before and after the integration with other parts.

Verification should be executed from bottom up to system. This is called *bottom up verification*, as shown in Figure 5.1. To reduce the verification time, testbenches should be generated concurrently (at the same time) with the (RTL) coding of functions.

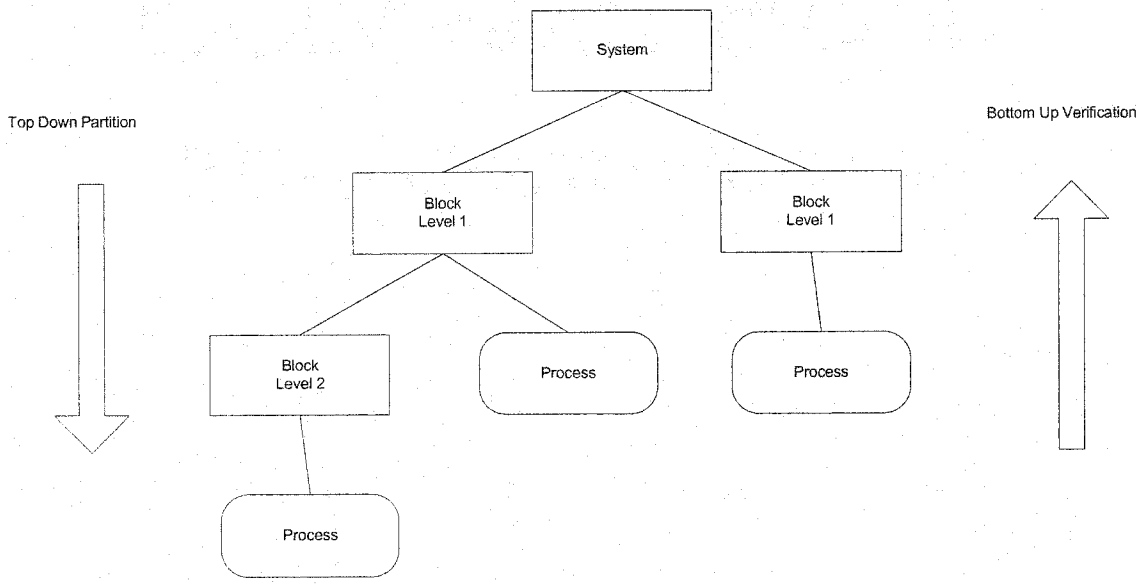



Figure 5.1: Top down design and bottom up verification

**3) Automation of the testbench generation.**

As we have already mentioned, automation of the testbench generation greatly relieves engineers from the burden of coding and debugging testbench programs.

## 5.3 RULE TECHNIQUE

### 5.3.1 Definition of a Rule

*Rules* tell which information is needed from the functional abstraction, and how to process this information to generate testbench. In other words, rules define the inputs, outputs and relation among them.

*Rules* are independent from any other system description languages such as SDL or SystemC. Rules focus on the relation between the functional abstraction and the testbench functional level. They define which information is needed to generate a reusable testbench, but are not limited to a specific method retrieving this information. That makes rules work with any abstraction methods.

*Rules* are first defined in English and could be implemented with any high-level software languages such as C++. *Rules* are the core of the tool, then, implementing the rules is developing the tool RTGT.

In Section 5.3.5, we will give an example of *Rules*.

### 5.3.2 Why Use Rules

We present three reasons:

1. **Easy to read as a design document**

*Rules* are written in English, and they clearly document the method to generate testbench step by step. We can understand the whole structure and the flow of the tool without the need to read the source code.

2. **Easy for future updating**

Here, updating the tool means adding or modifying rules. To do this, we need first to read the rule document and define the new rules that will be updated. All other rules stay the same.

3. **Flexible and software module reusability**

Rules are divided into three levels, and each level relates to a software module. Different rules can be combined together for different targets. The software modules perform as building blocks. These modules could be combined to build different testbenches. In this way, software modules can be reused during the tool development.

### 5.3.3 Rules Implementation

As mentioned in the previous section, we define *rules* at three levels. This facilitates the reusing of rules:

- High-level rules that focus on testbench partitioning, and testbench module generation for each partition.
- Middle-level rules that generate function interface for modules.
- Low-level rules that handle code generation details for each function.

High-level rules define how many modules the testbench needs for a specific DUT, without providing details of implementation. Accordingly, tool will generate a testbench

file for each module defined by rules. After, high-level rules are executed and middle-level rules will be called.

For each module, several middle-level rules are combined together to define which testbench methods should be included. Then, rules will generate the functional interface for those methods, and call low-level rules. As each rule only relates to one testbench method, we can easily find rules affected by an update.

Low-level rules represent the last step. For reasons of combination flexibility and easiness of modification, each low-level rule only performs a minimum of functions.

In the future, when rules will be updated, only affected sub-rules would be updated. Modification will be performed like a local update, without the necessity of rebuilding all the system.

### 5.3.4 Pattern

Rules have been defined to generate testbench components. However, not all parts are easy to generate fully automatically. Difficulties come from the fact that the *Parser* is not enough intelligent to retrieve all the necessary information from SDL specification. Also, we have defined a large set of rules able to deal with different situations. This is the role of the program to decide which rules should be called next. As certain combination of rules will often be used to deal with common case, we define *pattern* to automatically combine certain rules. Then, the process is speedup.

For a certain kind of designs, these patterns can automate the generation of the testbench that are tedious to automate by general rules. *Pattern* is a set of rules combined together in certain order to achieve a specific task. Each time, the tool needs to choose a pattern, rules are combined together to automate the testbench generation.

For example, different designs may have different emulation modules for DUT. First, the tool will define which emulation modules will be generated for this given DUT. These designs could be categorized in certain classes. Most of designs only need an *Injector* that injects stimuli into DUT, a *Monitor* that receives output from DUT and, if necessary, response to DUT. Then, this kind of designs can be classified into a pattern, *Injector-Monitor Pattern* (I-M Pattern) as shown in Figure 5.2. One example is PCI interface device, shown in Figure 3.7.

**Injector-Monitor Pattern (I-M Pattern)**

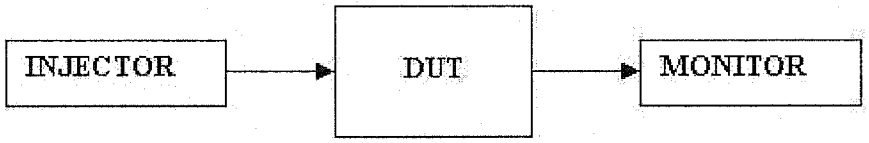



Figure 5.2: I-M Pattern

Here, the tool will automatically generate the *Injector* and *Monitor* modules.

For some other designs, not only *Injector* and *Monitor*, but also other modules are required. Considering the example of the ATM switch (Figure 4.8), a management module is required.

**Injector-Monitor Plus Pattern (I-MP Pattern)**

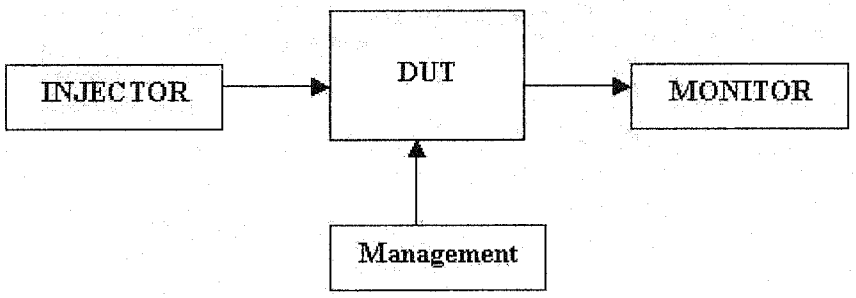



Figure 5.3 : I-MP Pattern

If we cannot foresee exactly which emulation module will be included besides *Injector* and *Monitor*, we will use the pattern Injector-Monitor Plus Pattern (I-MP Pattern) as presented in Figure 5.3, where the *Plus* must be defined by the user. In other words, the tool indicates the need of additional modules besides *Injector* and *Monitor*, and give the functional interface for these unknown modules. Then, the user will define these modules by given the specification and by adding programs.

Here:

- *Injector* and *Monitor* module will be automatically generated.
- From system description, the number and names of other emulation modules are defined.
- Each additional emulation module is generated.

Pattern can reduce program run-time by making most of routines be executed automatically. Then, the tool does not need to decide at each node of the flow, which rule will be called next.

With the study of different kinds of design, more patterns could be eventually defined to deal with different situations.

### 5.3.5 Rule Example

To present our methodology in Chapter 4, the ATM switch has been presented as a concrete example. Here, we use the same example to show an application of rules. Figure 5.4 shows the flow of rules.

First, the *Rule No.1* starts the whole process of the testbench generation. It is a high level rule to partition testbench into modules. The reason to do this partition is to give the testbench components reusability based on the theory of software reuse. The left column

of Figure 5.4 gives the rule reference number, while the next column provides the rule definition in English. Finally the third column provides software functions implementing this rule.

More precisely, *Rule No.1* defines a testbench that could be partitioned by the methodology APHVR into the following modules:

At the functional level:

- Verification Ring
- Emulation module
- Input Package
- Scoreboard
- Checker
- Coverage
- Glue Logic

At the low level:

- HDL Aspect
- Configuration Aspect (Test Case)

Except for the emulation module, all other modules are prefixed. That means whatever the DUT to be verified, all these modules are necessary. The difficulty for the emulation modules comes from the fact that functions needed to be emulated, vary from one DUT to another. Here, Rules should resolve this issue. Also, since we define a set of rules as *Pattern*, this becomes easier. We can predefine patterns, and each time only one decision need to be made by the rule i.e. which pattern will be used in this design.

| Rule Num | Rule Description | Related Implement Module |
|---|---|---|
| No.1<br>HL | Partition verification testbench. Define testbench module according to aspect-oriented methodology. Testbench is decomposed to following modules:<br>a. Verification Ring Call Rule No.1.1<br>**b. Emulation module Call Rule No.1.2**<br>c. Input Package Call Rule No.1.6<br>d. Scoreboard Call Rule No.1.7<br>e. Checker Call Rule No.1.8<br>f. Coverage Call Rule No.1.9<br>g. Glue Logic Call Rule No.1.10 | class TAG_Rule_Partition<br><br>Member function<br>a. Create_VerificationRing<br>**b. Create_Emulation.Modules**<br>c. Create_InputPackage<br>d. Create_Scoreboard<br>e. Create_Checker<br>f. Create_Coverage<br>g. Create_GlueLogic |

| | | |
|---|---|---|
| No. 1.2.1 | Define witch pattern used in this project. (Referenced document pattern. )<br><br>a. If I-M Pattern, call rules for generate injector and monitor.<br>Call Rule No.1.3 and Rule No.1.4<br>**b. If I-MP Pattern, call rules for generate injector, monitor and the rules for the other emulation module Call Rule No.1.3, Rule No.1.4 and Rule No.1.5**<br>c. If not belong any defined pattern. Call non pattern rules | Member function IMPattern() of class TAG_Rule_Partition<br><br>At a. call member function Create_InjectPort() Create_Monitor() of class TAG_Rule_Partition<br><br>**at b. call member function Create_InjectPort() Create_Monitor() GenerateFile(Other)** |

| | | |
|---|---|---|
| No.1.3.1 | Generate 'e' file for module injector<br>a. Get project name from interface<br>b. Combine project name with "_InjectPort.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call InjectPort RULES to add functions. Rule No.3 | Member function GenerateFile (InjectPort) of class TAG_Rule_Partition<br><br>At d. call member function CreatCode() of class TAG_Rule_InjectPort |

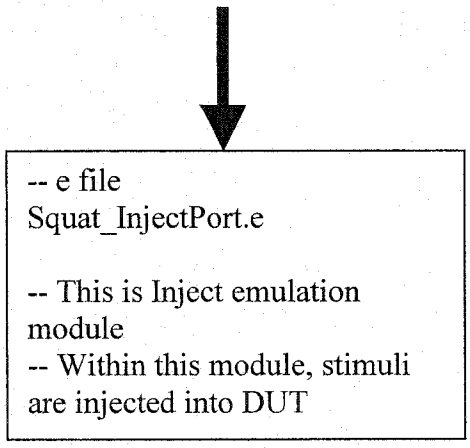



Figure 5.4: A concrete example shows rule flow

*Rule No.1* partitions the testbench and calls lower-level rules to perform further work. Rule *No.1.b*, is for emulation module definition. It calls sub-rule, *Rule No.1.2*, to define which emulation modules are to be used exactly.

Afterwards, *Rule No.1.2.1* becomes a sub-rule of *Rule No.1.2*, and the function of this rule is to determine the pattern. In our example shown in Figure 4.8, three emulation modules are needed: Injector, Monitor, and Management. Obviously, this belongs to *I-MP Pattern*. Then, *Rule No.1.2.1.b* is chosen. This rule tells that testbench need three emulation modules and calls sub-rules that specifically generate a *e* file to each module.

*Rule No.1.3.1* is specialized to the *injector* module generation. It also calls low-level rule to define functions of this module and to generate code.

At last, the output is a testbench module files written in *e* language, which represents what we expected for this tool.

## 5.4 DESIGN OF THE GENERATION TOOL SET RTGT

### 5.4.1 The Whole Picture of the Tool Set

The tool set RTGT is developed to assist testbench generation process based on our methodology introduced in Chapter 4. The flow of the tool is designed exactly following the methodology.

**Input:** SDL system description

**Output:** Functional reusable testbench modules

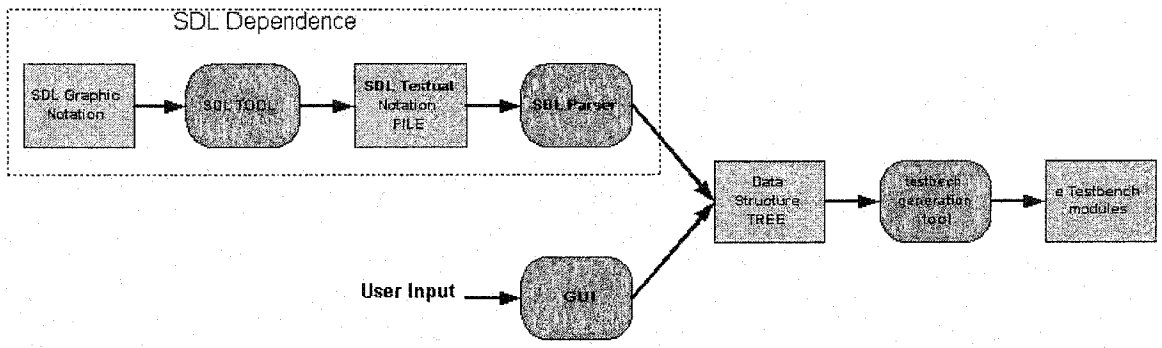



Figure 5.5: The whole picture of tool set RTGT

The tool takes a SDL specification as input and retrieves important information from this functional abstraction. At the same time, user could interact with a graphic interface. As shown in Figure 5.5, the RTGT are divided into three parts: 1) the retrieved data from the system abstraction that depends on one specific abstraction capturing language, i.e. SDL (see Chapter 4). 2) The GUI interface (Section 5.4.2) and the Automated Testbench Generator (ATG) (Section 5.4.3).

### 5.4.2 The Graphic User Interface (GUI)

As an alternative and an assistant of data retrieve process, a GUI has been developed to assure the user interact with the tool. Here, is the specification:

Inputs: User's inputs
Output: A data structure that contains information for the testbench generation.
Function: Collect the information from the graphic interface.

### 5.4.3 Automated Testbench Generator (ATG)

This is the core of the tool set RTGT. The ATG tool implements *rules* and generates testbench with the *e* language. Here, is the specification:

Inputs: Data retrieved from SDL Parser and GUI.
Outputs: Testbench modules coded by the *e* language.
Function: The automation of the testbench generation modules based on the retrieved data from the SDL parser and the GUI.

#### 5.4.3.1 Design Structure of ATG

Figure 5.6 shows the structure of the ATG and the relation with the *Parser* and the *GUI*. The tool is partitioned into several modules, which are implemented by *class*, to perform different functions.

An interface class restores information collected by the *Parser* and the *GUI*. All other classes within ATG share this information.

Class *ATG_rule_partition* partitions the testbench into seven modules. Then, we define software class specifically aiming to generate codes for each testbench module, such as class *ATG_rule_Gluelogic* for module *Gluelogic*, class *ATG_rule_scoreboard* for module *scoreboard*, etc.

In our research project, modules to develop can be chosen. This figure also shows how different works can be integrated together seamlessly. Each class is independent from each other and they share the data from class the ATG interface.

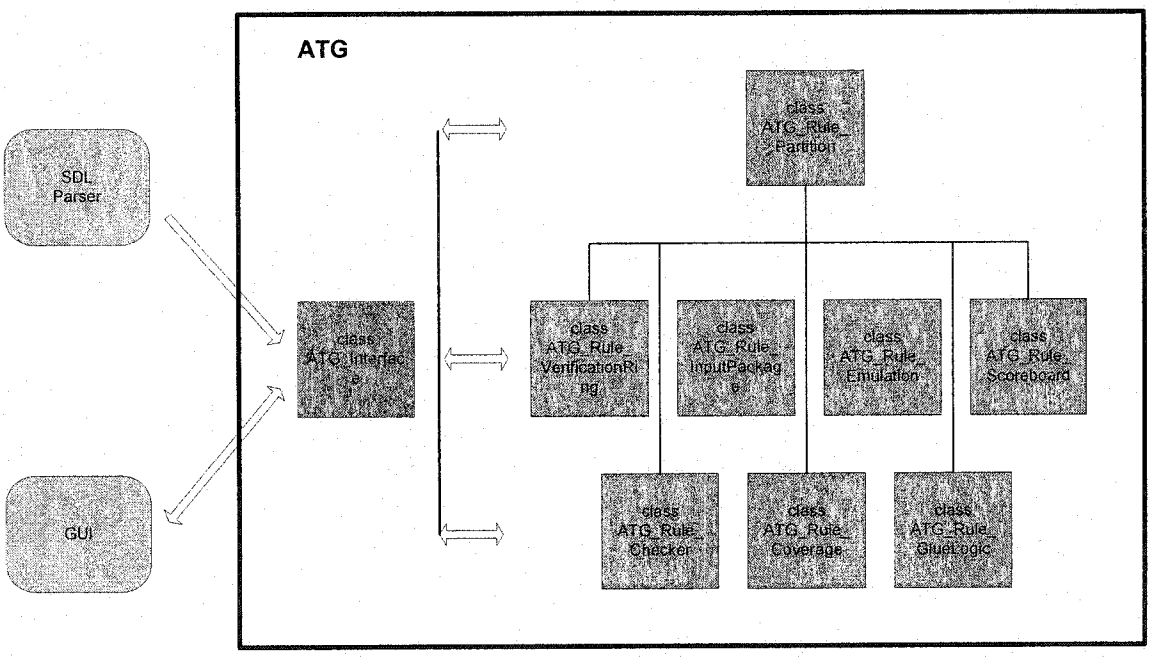



Figure 5.6: Structure of tool ATG

### 5.4.3.2 Implementation Rules in C++

The function of each software class is defined by *rules*. Then, developing software class is equal to implement *rules*.

The rules are implemented in C++:

- Each high level rule is implemented as a class
- Middle and low level rules are implemented as member functions.
- Each rule is assigned to one class or function.

Example 1 shows the C++ implementation of Rule No.1. This high level rule is implemented as a class TAG_Rule_Partition. The class contains instantiates objects of other class, and these classes are related to other software classes. The class TAG_Rule_Partition also defines some member functions, and for each function, it relates middle (or low) level rule.

**Example 1**

RULE No.1

```
class TAG_Rule_Partition { //HIGH LEVEL PARTITION RULE. RULE NO.1
private:
        TAG_Interface CInterface;
        TAG_Rule_VerificationRing CVRing;
        TAG_Rule_GlueLogic CGLogic;
        TAG_Rule_Scoreboard CSCBoard;
        TAG_Rule_InjectPort CInjePort;
        TAG_Rule_Monitor CMonitor;
        TAG_Rule_InputPackage CIPackage;
        TAG_Rule_Emulation CEmulation;

public:

//****** HIGH LEVEL RULE NO1 DEFINE MIDDLE LEVEL RULES
int Create_VerificationRing(); //Create testbench module -- verification ring
                               //MIDDLE LEVEL RULE. RULE NO.1.1
int Create_EmulationModules(); //Create testbench emulation modules
                               //MIDDLE LEVEL RULE. RULE NO.1.2
int Create_InputPackage();    //Create testbench InputPackage module //RULE NO.1.6
int Create_Scoreboard();      //Create testbench scoreboard module   //RULE NO.1.7
int Create_Coverage();        //Create testbench coverage module     //RULE NO.1.8
int Create_Checker();         //Create testbench assertion checker module //RULE NO.1.9
int Create_GlueLogic();       //Create testbench glue logic module   //RULE NO.1.10

//******************  END OF HIGH LEVEL RULE DEFINE
```

```
//****** MIDDLE LEVEL RULES DEFINE MIDDLE LEVEL RULES
int IMPackage(); //deside which package used for emulation module
                    //MIDDLE LEVEL RULE //RULE NO.1.2.1
int Create_InjectPort();      //Create testbench InjectPort module //RULE NO.1.3
int Create_Monitor();         //Create testbench Monitor module    //RULE NO.1.4

//******************** END OF MIDDLE LEVEL RULE DEFINE

//****** LOW LEVEL RULES DEFINE LOW LEVEL RULES
int GenerateFile(Module Mname); //Generate e file for each modules
                              //LOW LEVEL RULE //RULE NO.1.1.1
//******************** END OF LOW LEVEL RULE DEFINE
};
```

## 5.5 LIMITATION OF THE APPROACH

In this section we raise some questions related to the features of the tool.

### 5.5.1 How Complete is the Automatic Testbench Generator

The testbench generated by the tool are not complete. In this section, we present which parts are either done or not done automatically.

The discussion has two levels of meaning:

- Some parts of the testbench have not been completely generated by the tool because relative software classes have not been completed. Those parts of testbench could be completed in the future work.
- Some parts of testbench have not been completely generated by the tool because they are too much custom to be automatically generated.

As we mentioned, according to APHVR methodology, a complete testbench can be divided into 9 modules (see Section 5.3.5).

Tool does not generate all low-level testbench modules (e.g. module *HDL Aspect* and module *Configuration Aspect*), because the tool generate reusable testbench component at high-level (e.g. the functional level).

The frame of the high-level testbench modules is automatically generated. The frame includes all modules file, the interface for necessary functions within the module, and a *Glue Logic* module that connects all modules together.

Each module could be developed independently. We have targeted some modules in this project to develop methods to completely generate the testbench code. The modules not chosen during the first step remain uncompleted. It belongs to the first meaning of "completion" we mentioned above, and could be completed in the future.

If we take the *Scoreboard* module as an example, it compares output data with the value they are supposed to be. The *Scoreboard* (see Figure 2.5) module needs to monitor output data, and do comparison with a reference model. In this module, functions like *add transaction* or *match* are completely generated by the tool, but there is a function that performs "mapping" between data collected by the scoreboard and the reference module. We have decided to let to the user the definition of this function.

Indeed, this *reference module* changes each time we verify a new DUT. This implies automatic generation of the *mapping* module very complicated, because there is no curtain format for these *reference modules* and there is no way in the tool to handle these manual work without a formal writing format.

On the other hand, manually writing this "mapping" program is not a difficult work. The function of this program is to compare certain data that are defined in the *reference module.* Then, the program is mainly combined by *if* statements, as shown by in example 2. Programmers even not very familiar with the *e* language could finish it very soon.

**Example 2**

```
-- we compare the fields of the cell
if t.check_crc() == FALSE then {
  print "Error";
  print me;
  dut_error("Bad CRC");
};
if source.header.vci != t.header.vci or          -- mapping
  source.header.clp != t.header.clp or
  source.header.pt  != t.header.pt then {
  print "Error";
  print me;
  dut_error("Corruption on unmodified fields:");
};
```

Then, after considering this tradeoff, we have decided to let the user to finish these parts of testbenches. We also provide *patterns* for the part that is tedious to generate automatically.

### 5.5.2 The Difficulty for User to Finish Incomplete Testbench

Sometimes when a program is not completed, programmer who achieve the work, struggle in figuring out program structure, and then spend less time by rewriting from scratch.

Thus, a critical question exists: since testbench being built by the tool is not complete, will users get lost in this generated code? The answer is no. During the design of the tool, we have already realized this issue and find the solution to avoid this problem.

First, the testbench being generated by the tool is decomposed into modules with proper method. Module is combined with functions. These functions have the basic functionality and are easy to be understood. Well-defined testbench assure no difficulty to figure out the functionality of incomplete function. A module named *Glue Logic*, which is

completed, connects all testbench modules together. This tool could generate the whole frame of testbench: modules files, function interfaces, connections, and most of codes. Then, engineers only need to focus on incomplete functions.

Secondly, when the tool generates testbenches, it will generate also documentation. The tool provides to the user a report that shows the whole structure and modules, and tell to the user which parts has been done and which parts need to be defined manually. Then, the user can understand the whole testbench.

Third, comments are added in the testbench program during the tool generation. It is easy for the user to read source codes and, more importantly, to give information (a sort of guide) for the missing part.

### 5.6 Analysis of the ATG Tool

In this section we analysis the performance of ATG. Two examples have been tested: the ATM Switch (Section 4.2.3.1) and a Memory Manager (Section 3.2.2.1).

Before we present results, the following points must be considered:

- ATG at this stage can only be efficiently used for DUTs that belong to pattern I-M and I-MP (Section 5.3.4)
- Testbenches comparisons between fully manually generated and generated by the ATG tool are all based on the APHVR methodology (Section 3.2.2.1) and our reusable testbench methodology approach (Chapter 4).
- In this project, three functional level testbench modules (a total of seven modules, Section 5.3.5) have been developed to generate ATG. We analysis these three modules.

Table 5.1 showed codes (fully) manually generated and generated by the ATG tool with the ATM Switch example, while table 5.2 showed the example for the Memory Manager.

| Testbench Module | Lines of Code Generated Manually | Lines of Code Generated by ATG | Percentage of Completion |
|---|---|---|---|
| **Verification Ring** | 98 | 98 | 100% |
| **Glue Logic** | 152 | 132 | 86% |
| **Scoreboard** | 169 | 125 | 74% |

Table 5.1: Performance analysis with the ATM Switch

| Testbench Module | Lines of Code Generated Manually | Lines of Code Generated by ATG | Percentage of Completion |
|---|---|---|---|
| **Verification Ring** | 129 | 129 | 100% |
| **Glue Logic** | 167 | 141 | 84% |
| **Scoreboard** | 250 | 130 | 52% |

Table 5.2: Performance analysis with the Memory Manager

| Example | Total Lines Generated Manually | Total Lines Generated by ATG | Average Percentage of Completion |
|---|---|---|---|
| **ATM Switch** | 419 | 355 | 85% |
| **Memory Manager** | 546 | 400 | 73% |
| **Average Performance of Two Examples** | 965 | 755 | 78% |

Table 5.3: Average percentage of completion

**Analysis:**

- As shown in Table 5.3, the ATG tool has done 85% of the coding task of these three modules with the DUT ATM Switch and 73% with the DUT Memory Manger. ATG can achieve, in average, 78% of completion for these two examples.
- In Table 5.3, 78% of the examples are generated automatically in average. Then, the design is speedup for this part of the testbench, since programming and debugging time is gained. For certain cases, debugging time could be two times of the coding time.
- The percentage of uncompleted programs is 22% and these programs have been completed manually. As we have discussed in Section 5.5.2, it is not hard for verification engineer to complete the program. Indeed, using the same programming methodology (APHVR), engineers understand the partitioning and the structure of testbench. This helps them to know which part need to be completed manually and how to perform it.
- For the module *Glue Logic* of the ATM Switch, 14% is uncompleted. Similarly, 16% is uncompleted for the Memory Manager. This is due to the pattern I-MP. These two examples all belong to the *Injector-Monitor Plus* Pattern. As we have mentioned in Section 5.3.4, beside Injector and Monitor, another module is unknown. The ATG tool can generate a functional interface for the *Glue Logic* module, but cannot provide function for the unknown module. This task needs to be done manually. For the DUTs belong to *Injector-Monitor* pattern, module *Glue Logic* could be completely generated.
- For the *Scoreboard* module of the ATM Switch discussed in detail in Section 5.5.1, 74% has been done by the ATG tool, but only 52% for the Memory Manager. The reason is, for the DUT Memory Manager, two scoreboards need to be generated (reference Figure 3.4), while only one is necessary for the DUT ATM Switch. This illustrates, why the performance of the ATG has dropped. A

way to solve this problem is to add more rules and patterns to deal with different situations.

**Conclusion:**

In average, the ATG tool has done 78% of the coding work. If we consider the debugging time that is gained, ATG is very helpful for verification engineer to generate testbenches. Although at this stage, ATG can only work efficiently with relative simple DUTs. However, ATG has a promising future (see the discussion in Section 5.5), if we could eventually include other class of designs and develop more rules..

## 5.7 SUMMARY

In this chapter we first introduce *rule-based* technique. Also, through an example, we have discussed what are *rules*, why use *rules*, how the *rules* work. Finally, we have presented patterns for parts of the testbench that are hard to be automatically generated.

Then, we have presented the design of the tool. We showed structure of tool sets and the functionality of each tool. Some program examples have been presented to demonstrate the software implementation of *rules*.

We also raise some critical questions related to the tool during the design and give the solutions.

# CHAPTER 6
# CONCLUSION

In this project, we proposed a methodology to build reusable testbenches at the functional abstraction level, and a prototype tool has been developed. More precisely, we have presented a methodology to capture system functional specification, and a methodology to build reusable testbenches with the information retrieved from system functionalities. Also we have proposed the concept of rule-based verification to automate testbench creation. The thesis presents a methodology to do functional verification with this tool.

We have introduced a method that uses the System Description Language (SDL) to capture system functional specification. SDL has several advantages. First it supports both graphic presentation and textual notation. The graphic presentation makes it easy to describe a system and the textual notation can be used as a standard input to an automatic testbench generation tool. Second, SDL is an executable language, thus specifications written with SDL could be verified before being implemented. This is very important as the specifications define what the right system is supposed to be.

A new verification reuse methodology has been developed in this project that uses SDL to describe system specifications and build testbenches at high level. The same specification could have different implementations. However, these implementations share common functionalities at the functional level defined by the specification. We have captured these common functionalities with SDL and generated testbenches accordingly at the functional level. Such testbench is used to verify common behavior among all implementations. This can obviously be reused in different projects.

We also have developed a tool named RTGT to implement this methodology. This tool has been designed with the *rule-based* technique in order to facilitate future expansion and to make the tool more flexible. This RTGT tool include a *parser* to retrieve

information from SDL system specification, a GUI for convenient user's interaction, and the ATG tool to automate the generation of testbenches with *e* language based on our verification reuse methodology.

According to the APHVR methodology, functional level testbench can be divided into 7 modules (reference Section 5.3.5). Accordingly, the ATG tool is divided into several parts. Each part specifically focuses on one class of testbench modules. Rules has been completed for certain parts, but others have to be completed in the future:

**Glue Logic**

The rules for this module have been defined and implemented

**Verification Ring**

The rules for this module have been defined and implemented.

**Scoreboard**

The rules for this module have been defined and implemented.

**Emulation module**

High-level rules have been defined and implemented. They include:

- According to an SDL specification, define how many and which emulation modules need to be generated. A pattern is defined to reduce the user's work.
- For the emulation modules within the patterns, such as, injector or monitor, necessary frames are generated.

**Input Package**

High-level rules have been defined and implemented. They include:

- Module file generation

- Generation of necessary structures of the functional interface

**Checker**

Should be developed in the future.

**Coverage**

Should be developed in the future.

At this stage, the tool can automate the generation of testbench modules. With this tool:

- Testbench decomposition is completely done.
- All the necessary modules are defined and are given functional interfaces.
- Most necessary functions within each modules are defined and some of them come with the e code
- The blank functions are given frame and commented to tell what the functionality should be.
- Class extension, class instantiation, and packet injection are given in the glue logic module. This module links entire modules together.
- The testbench code generated by this tool can be compiled without error.

To complete the testbenches, the verification engineer must

- Understand the whole structure.
- To make clear what has been done and what needs to be added into the testbench.
- Fill code into blank functions of the modules that are incomplete.

Based on the analysis and results reported above, the following projects can be made to further improve the performance of the ATG.

- For the *emulation module* and *input package,* the work must be completed. Since we only defined high-level rules for these two modules, more work is needed in defining detailed rules for automatic generation. Also there is a need to define which informations are required and whether they should come from the SDL parser or the GUI. The *parser* and *GUI* may need to be updated accordingly. Then, the implementation of these rules and their integration into the tool must be completed.

- Also, more patterns should be defined. We have introduced patterns to handle the parts that are hard to automate. We have defined a pattern for a specific design to automate the generation of specific parts of testbench programs. It is necessary to study different designs and to define other patterns.

- Finally, The GUI could be expanded to give users an easy-mastered interface to this tool. And at the same time, we could also try to retrieve as much information as possible from the parser to reduce human intervention during testbench generation.

## BIBLIOGRAPHY

[1] BERGERON, J., "Writing Testbenches, Functional Verification of HDL Models", Kluwer Academic Publishers, 2000

[2] École Polytechnique de Montréal, "Functional Verification Survey [Web Page]". http://www.grm.polymtl.ca/~lemire/survey

[3] KIRSHENBAUM, Z., "Verification Reuse Offers Real Benefits", EEDesign, March 6, 2001.

[4] KUNJAN, T., BOHN, T., "A Practicable Test Environment for Reuse Purposes", http://www.infotech.tu-chemnitz.de/~microtec/eng/press/annualrep98/reuse.htm

[5] SFORZA, F., BATTU, L., BRUNELLI, M., CASTELNUOVO, A., MAGNAGHI, M.A., "Design for Verification", Methodology Quality Electronic Design, 2001 International Symposium on, 2001 Page(s): 50-55.

[6] FISCHER, J., HOLZ, E., MOLLER-PEDERSEN, B., "Structural and Behavioral Decomposition in Object Oriented Models", (ISORC 2000) Proceedings, Third IEEE International Symposium, 2000, Page(s): 368 --375

[7] IONTA, R., "Verification Ensures Reuse Really Used", EETimes, December 22, 2000.

[8] REGIMBAL, S., LEMIRE, J.F., SAVARIA, Y., BOIS, G., ABOULHAMID, E.M., BARON, A., "Aspect Partitioning for Hardware Verification Reuse", IWSOC2002, 2002, pp.

[9] http://www.verisity.com

[10] MONTLICK, T., "What is Object-Oriented Software?", Software Design Consultants, LLC, http://softwaredesign.com/objects.html

[11] http://www.open-vera.com/

[12] MAVADDAT, F., "Formal Hardware Verification: A Users' View Summary", ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International, 1998, Page(s): 418 –418

[13] KUMAR, R., "Formal Verification of Hardware: Misconception and Reality", Wescon/98, 1998, Page(s): 135 –138

[14] ELLEITHY, K.M., AREF, M.A., "A Rule-Based Approach for High Speed Adders Design Verification", Proceedings of the 37th Midwest Symposium, Circuits and Systems, 1994, Volume: 1, 1994, Page(s): 274 -277 vol.1

[15] FERRANDI, F., RENDINE, M., SCIUTO, D., "Functional Verification for SystemC Descriptions Using Constraint Solving", Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, 2002, Page(s): 744 –751

[16] ROY, S.K., RAMESH, S., "Functional Verification of System on Chips - Practices, Issues and Challenges", Design Automation Conference, 2002, Proceedings of ASP-DAC 2002, 7th Asia and South Pacific and the 15th International Conference on VLSI Design, Proceedings, 2002, Page(s): 11 –13

[17] SFORZA, F., BATTU, L., BRUNELLI, M., CASTELNUOVO, A., MAGNAGHI, M.A., "Design for Verification Methodology", Quality Electronic Design, 2001 International Symposium on, 2001, Page(s): 50 –55

[18] LIU, C., CHEN, I.L., JOU, J.Y., "An Efficient Design-for-Verification Technique for HDLs", Proceedings of the ASP-DAC 2001. Design Automation Conference, 2001. Asia and South Pacific, 2001, Page(s): 103 –108

[19] EVANS, A., SILBURT, A., VRCKOVNIK, G., BROWN, T., DUFRESNE, M., HALL, G., TUNG, H., LIU, Y., "Functional Verification of Large ASICs", Design Automation Conference, 1998, Proceedings, 1998, Page(s): 650 –655

[20] PELLERIN, D., TAYLOR, D., "VHDL Made Easy!", Prentice Hall PTR, 1997

[21] Specman Elite Online Documentation 4.0.2, Verisity Design, Inc., 2002

[22] NSAME, P.A., IBM PCI-X Hard IP Core Bus Protocol & Data Integrity Checking, IBM Microelectronic Division, 2000

[23] PCI Local Bus Specification, Revision 2.2, PCI-SIG, December 18, 1998

[24] PCI-X Addendum to the PCI Local Bus Specification, Revision 1.0a, PCI-SIG, Jul. 24, 2000

[25] "Coverage-Driven Functional Verification, Using Coverage to Speed Verification and Ensure Completeness", Verisity Design, Inc. 2001

[26] HOLLANDER, Y., MORLEY, M., NOY, A., "The e Language: A Fresh Separation of Concerns", Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38. Proceedings, 2001, Page(s): 41 –50

[27] BERGERON, J., "Languages Run Verification Ecosystem", EETime, October 16, 2000

[28] http://www.testbuilder.net/

[29] DOLDI, L., "Visually Design Executable Models", SDL Illustrated, 2001

[30] CCITT, COM X-R, 17-E, Geneva, March 1992, "Recommendation Z.100 – CCITT Specification and Description Language (SDL) and Annex A to the Recommendation"

[31] BELINA, F., HOGREFE, D., "The CCITT Specification and Description Language SDL", Networks and ISDN system, 16, pp. 311-341, North-Holland, 1988/89

[32] http://www.systemc.org

[33] http://www.sdl-forum.org/sdl88tutorial/

[34] http://www.janick.bergeron.com/

[35] http://www.telelogic.com/products/tau/sdl/index.cfm

[36] GOERING, R., "Synopsys Opens Vera as Verification Language Standard", EE Time, April 4, 2001

[37] VAHID, F., GIVARGIS, T., "Embedded System Design, A Unified Hardware/Software Introduction", John Wiley & Sons, Inc. 2002

[38] GERLACH, J., ROSENSTIEL, W., "System Level Design Using the SystemC Modeling Platform (SDL 2000) ", http://www.systemc.org/projects/sitedocs/document/sda-2000/en/1

[39] VANDERPERREN, Y., SONCK, G., OSTENDE, P.V., PAUWELS, M., DEHAENE, W., MOORE, T., "A Design Methodology for the Development of a Complex System-On-Chip Using UML and Executable System Models" http://www.systemc.org/projects/sitedocs/document/DesignMethodologyUML_SystemC/en/1

[40] NSAME, P.A., "IBM PCI-X Hard IP Core Bus Protocol & Data Integrity Checking", IBM Microelectronic, Aug.31, 2000

## APPENDIX A: EXAMPLE OF RULES TABLE

The Following is part of rules table, which includes rule number 1 and its' sub rules.

| Rule Num | Rule Description | Related Implement Module |
|---|---|---|
| No.1<br>HL | Partition verification testbench. Define testbench module according to aspect-oriented methodology.<br>Testbench is decomposed to following modules:<br>h. Verification Ring Call Rule No.1.1<br>i. Emulation module Call Rule No.1.2<br>j. Input Package Call Rule No.1.6<br>k. Scoreboard Call Rule No.1.7<br>l. Checker Call Rule No.1.8<br>m. Coverage Call Rule No.1.9<br>n. Glue Logic Call Rule No.1.10 | class<br>TAG_Rule_Partition<br><br>Member function<br>h. Create_Verification Ring<br>i. Create_EmulationModules<br>j. Create_InputPackage<br>k. Create_Scoreboard<br>l. Create_Checker<br>m. Create_Coverage<br>n. Create_GlueLogic |
| No.1.1<br>ML | Generate Module Verification Ring.<br>a. Call level rule with proper parameter ""<br>Call Rule No. 1.1.1 | Member function<br>Create_VerificationRing<br>()<br>of class<br>TAG_Rule_Partition |
| | | call member function<br>GenerateFile<br>(Module Mname)<br>of class<br>TAG_Rule_Partition |
| No.1.1<br>.1<br>LL | Generate 'e' file for module verification ring<br>a. Get project name from interface<br>b. Combine project name with "_VRing.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Verification RULES to add functions. RULE No.2 | Member function<br>GenerateFile<br>()<br>of class<br>TAG_Rule_Partition |
| | | at d. call member function<br>CreatCode.ofstream& file)<br>of class<br>TAG_Rule_Verification Ring |

| | | |
|---|---|---|
| No.1.2 ML | Generate Module Emulation<br>a. Call low level rule No.1.2.1 to define which package used | Member function Create_EmulationModules () of class TAG_Rule_Partition |
| | | at a. call member function IMPackage() of class TAG_Rule_Partition |
| No. 1.2.1 LL | Define witch package used in this project. (Referenced document package. )<br>d. If I-M Package, call rules for generate injector and monitor.<br>Call Rule No.1.3 and Rule No.1.4<br>e. If I-MP Package, call rules for generate injector, monitor and the rules for the other emulation module<br>Call Rule No.1.3, Rule No.1.4 and Rule No.1.5<br>f. If not belong any defined package. Call non package rules | Member function IMPackage() of class TAG_Rule_Partition |
| | | At a. call member function Create_InjectPort() Create_Monitor() of class TAG_Rule_Partition<br><br>at b. call member function Create_InjectPort() Create_Monitor() GenerateFile(Other) of class TAG_Rule_Partition |
| No.1.3 | Generate module for Injector<br>a. Call level rule with proper parameter "Injector"<br>Call Rule No. 1.3.1 | Member function Create_InjectPort () of class TAG_Rule_Partition |
| | | At a. call member function GenerateFile(InjectPort) of class TAG_Rule_Partition |
| No.1.3 .1 | Generate 'e' file for module injector<br>e. Get project name from interface<br>f. Combine project name with "_InjectPort.e" to generate correct file name<br>g. Create file with file name generated at b<br>h. Call InjectPort RULES to add functions. Rule No.3 | Member function GenerateFile (InjectPort) of class TAG_Rule_Partition |
| | | At d. call member function CreatCode() of class TAG_Rule_InjectPort |

| | | |
|---|---|---|
| No.1.4 | Generate module for Injector<br>a. Call level rule with proper parameter "Monitor"<br>Call Rule No. 1.4.1 | Member function<br>Create_Monitor ()<br>of class<br>TAG_Rule_Partition |
| | | At a. call member function<br>GenerateFile(Monitor)<br>of class<br>TAG_Rule_Partition |
| No.1.4<br>.1 | Generate 'e' file for module injector<br>a. Get project name from interface<br>b. Combine project name with "_Monitor.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Monitor RULES to add functions. Rule No.4 | Member function<br>GenerateFile<br>(Monitor)<br>of class<br>TAG_Rule_Partition |
| | | At d. call member function<br>CreatCode() of class<br>TAG_Rule_Monitor |
| No.1.5 | Generate module for 'other' modules of I-MP Package<br>a. Get module name form interface<br>b. Combine module name with ".e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Emulation RULES to add functions. Rule No.5<br>e. Repeat a~e if has other modules | Member function<br>GenerateFile(Other)<br>of class<br>TAG_Rule_Partition |
| | | At d. call member function<br>CreatCode() of class<br>TAG_Rule_Emulation |
| No.1.6 | Generate Module Input Package<br>a. Call level rule with proper parameter "InputPackage"<br>Call Rule No. 1.6.1 | Member function<br>Create_InputPackage()<br>of class<br>TAG_Rule_Partition |
| | | call member function<br>GenerateFile<br>(Module InputPackage)<br>of class<br>TAG_Rule_Partition |
| No.1.6<br>.1 | Generate 'e' file for module Input Package<br>a. Get project name from interface<br>b. Combine project name with "_IPackage.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Input Package RULES to add functions. RULE No.6 | Member function<br>GenerateFile<br>(InputPackage)<br>of class<br>TAG_Rule_Partition |
| | | at d. call member function<br>CreatCode(ofstream& file)<br>of class<br>TAG_Rule_InputPackage |

| | | |
|---|---|---|
| No.1.7 | Generate Module Scoreboard<br>a. Call level rule with proper parameter "Scoreboard"<br>Call Rule No. 1.7.1 | Member function<br>Create_Scoreboard()<br>of class<br>TAG_Rule_Partition |
| | | call member function<br>GenerateFile<br>(Module Scoreboard)<br>of class<br>TAG_Rule_Partition |
| No.1.7<br>.1 | Generate 'e' file for module Scoreboard<br>a. Get project name from interface<br>b. Combine project name with "_Scoreboard.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Scoreboard RULES to add functions. RULE No.7 | Member function<br>GenerateFile<br>(Scoreboard)<br>of class<br>TAG_Rule_Partition |
| | | at d. call member function<br>CreatCode(ofstream& file)<br>of class<br>TAG_Rule_Scoreboard |
| No.1.8 | Generate Module Checker<br>a. Call level rule with proper parameter "Checker"<br>Call Rule No. 1.8.1 | Member function<br>Create_Checker()<br>of class<br>TAG_Rule_Partition |
| | | call member function<br>GenerateFile<br>(Module Checker)<br>of class<br>TAG_Rule_Partition |
| No.1.8<br>.1 | Generate 'e' file for module Checker<br>a. Get project name from interface<br>b. Combine project name with "_Checker.e" to generate correct file name<br>c. Create file with file name generated at b<br>d. Call Checker RULES to add functions. RULE No.8 | Member function<br>GenerateFile<br>(Checker)<br>of class<br>TAG_Rule_Partition |
| | | at d. call member function<br>CreatCode(ofstream& file)<br>of class<br>TAG_Rule_Checker |
| No.1.9 | Generate Module Coverage<br>a. Call level rule with proper parameter "Coverage"<br>Call Rule No. 1.9.1 | Member function<br>Create_Coverage ()<br>of class<br>TAG_Rule_Partition |

| | | |
|---|---|---|
| | | call member function GenerateFile (Module Coverage) of class TAG_Rule_Partition |
| No.1.9 .1 | Generate 'e' file for module Coverage<br>e. Get project name from interface<br>f. Combine project name with "_Coverage.e" to generate correct file name<br>g. Create file with file name generated at b<br>h. Call Coverage RULES to add functions. RULE No.9 | Member function GenerateFile (Coverage) of class TAG_Rule_Partition |
| | | at d. call member function CreatCode(ofstream& file) of class TAG_Rule_Coverage |
| No.1.1 0 | Generate Module GlueLogic<br>a. Call level rule with proper parameter "GlueLogic"<br>Call Rule No. 1.10.1 | Member function Create_GlueLogic () of class TAG_Rule_Partition |
| | | call member function GenerateFile (Module GlueLogic) of class TAG_Rule_Partition |
| No.1.1 0.1 | Generate 'e' file for module GlueLogic<br>i. Get project name from interface<br>j. Combine project name with "_GlueLogic.e" to generate correct file name<br>k. Create file with file name generated at b<br>l. Call GlueLogic RULES to add functions. RULE No.10 | Member function GenerateFile (GlueLogic) of class TAG_Rule_Partition |
| | | at d. call member function CreatCode(ofstream& file) of class TAG_Rule_GlueLogic |
| | | |

# APPENDIX B: EXAMPLE CODE FOR ATG

## F.1. File Rule_Partition.h

```
////////////////////////////////////////////////////////////
//   Filename: Rule_Partition.h
//   Project:  ATG
//
//   Author:   Jiahong Wang
//
//   Description:define class TAG_Rule_Partition.It's the
//               high level rule of RULE NO.1
//
////////////////////////////////////////////////////////////


//#include "Interface.h"
#include "Interface.h"
//#include "Global.h"
#include <iostream>
#include <stdio.h>

#include "Rule_VerificationRing.h"
#include "Rule_GlueLogic.h"
#include "Rule_Scoreboard.h"
#include "Rule_InjectPort.h"
#include "Rule_Monitor.h"
#include "Rule_InputPackage.h"
#include "Rule_Emulation.h"
#include "Rule_Checker.h"
#include "Rule_Coverage.h"
using namespace std;

class TAG_Rule_Partition { //HIGH LEVEL PARTITION RULE. RULE NO.1
private:
     TAG_Interface* CInterface;
     TAG_Rule_VerificationRing CVRing;
     TAG_Rule_GlueLogic CGLogic;
     TAG_Rule_Scoreboard CSCBoard;
     TAG_Rule_InjectPort CInjePort;
     TAG_Rule_Monitor CMonitor;
     TAG_Rule_InputPackage CIPackage;
     TAG_Rule_Emulation CEmulation;
     TAG_Rule_Checker CChecker;
     TAG_Rule_Coverage CCoverage;

     enum Module
     {
          VerificationRing,
          InputPackage,
          Scoreboard,
```

```
        Coverage,
        Checker,
        GlueLogic,
        InjectPort,
        Monitor,
        Other};

public:

    void init(TAG_Interface& Interface);


//  TAG_Rule_Partition(TAG_Interface Interface);
    //*************************************************************
    //**** HIGH LEVEL RULE NO1 DEFINE MIDDLE LEVEL RULES ***//
    int Create_VerificationRing(); //Create testbench module -
                                    //verification ring
                                    //MIDDLE LEVEL RULE. RULE NO.1.1

    int Create_EmulationModules(); //Create testbench emulation
                                    //modules
                                   //MIDDLE LEVEL RULE. RULE NO.1.2
    int Create_InputPackage();     //Create testbench InputPackage
                                    //module //RULE NO.1.6
    int Create_Scoreboard();       //Create testbench scoreboard
                                    //module   //RULE NO.1.7
    int Create_Coverage();         //Create testbench coverage
                                    //module     //RULE NO.1.8
    int Create_Checker();          //Create testbench assertion
                                    //checker module  //RULE NO.1.9
    int Create_GlueLogic();        //Create testbench glue logic
                                    //module   //RULE NO.1.10

    //****  END OF HIGH LEVEL RULE DEFINE  ****//
    //*************************************************************

    //*********************************************************//
    //****** MIDDLE LEVEL RULES DEFINE MIDDLE LEVEL RULES ******//
    int IMPackage(); //deside which package used for emulation
                     //module
                    //MIDDLE LEVEL RULE //RULE NO.1.2.1


    int Create_InjectPort();       //Create testbench InjectPort
                                    //module //RULE NO.1.3
     int Create_Monitor();          //Create testbench Monitor module
                                    //RULE NO.1.4



    //******  END OF MIDDLE LEVEL RULE DEFINE  ***************//
    //*************************************************//
```

```
    //*****************************************************//
    //****** LOW LEVEL RULES DEFINE LOW LEVEL RULES **********//
    int GenerateFile(Module Mname);  //Generate e file for each
                                      //modules
                                //LOW LEVEL RULE
                                      //RULE NO.1.1.1


    //*************  END OF LOW LEVEL RULE DEFINE  ************//
    //*************************************************************//

};
```

## F.2. File Rule_Partition.cpp

```
////////////////////////////////////////////////////////////
//   Filename: Rule_Partition.cpp
//   Project:  ATG
//
//   Author:   Jiahong Wang
//
//   Description:implement class TAG_Rule_Partition.It's the
//               high level rule of RULE NO.1
//
////////////////////////////////////////////////////////////

#if defined(_MSC_VER) && (_MSC_VER > 1100)
    //Disable MS VC 6.0 warning 4786
    #pragma warning(disable:4786)
#endif

#include "Rule_Partition.h"
#include <fstream>
#include <iterator>
using namespace std;



void TAG_Rule_Partition::init(TAG_Interface& Interface)
{
    CInterface = &Interface;

}

int TAG_Rule_Partition::Create_VerificationRing() //RULE No. 1.1
{

    if(GenerateFile(VerificationRing)) // Call low level partition
                                       // rules to generate file
        return 0;
```

```
    else
        return 1;
}


int TAG_Rule_Partition::Create_EmulationModules()
{

    if(IMPackage())  //Call middle level rule to deside which package
used for emulation module
        return 0;
    else
        return 1;

}

int TAG_Rule_Partition::Create_InputPackage()
{
    if(GenerateFile(InputPackage)) // Call low level partition rules
                                   // to generate file
        return 0;
    else
        return 1;

}


int TAG_Rule_Partition::Create_InjectPort()  //Rule No. 1.3
{
    if(GenerateFile(InjectPort)) // Call low level partition rules
                                 // to generate file
        return 0;
    else
        return 1;
}

int TAG_Rule_Partition::Create_Monitor()  //Rule No. 1.4
{
    if(GenerateFile(Monitor)) // Call low level partition rules to
                              // generate file
        return 0;
    else
        return 1;
}

int TAG_Rule_Partition::Create_Scoreboard()
{
    if(GenerateFile(Scoreboard)) // Call low level partition rules
                                 // to generate file
        return 0;
    else
        return 1;
}
```

```
int TAG_Rule_Partition::Create_Checker()
{
     if(GenerateFile(Checker)) // Call low level partition rules to
                               // generate file
          return 0;
     else
          return 1;
}


int TAG_Rule_Partition::Create_Coverage()
{
     if(GenerateFile(Coverage)) // Call low level partition rules to
                                // generate file
          return 0;
     else
          return 1;
}

int TAG_Rule_Partition::Create_GlueLogic()
{
     if(GenerateFile(GlueLogic)) // Call low level partition rules to
                                 // generate file
          return 0;
     else
          return 1;
}



int TAG_Rule_Partition::GenerateFile(Module Mname)
{
     string filename, tmp;
     ofstream fstream;



     switch (Mname)
     {
     case VerificationRing:   // RULE No.1.1.1
          filename = (CInterface->Get_ProjectName() += "_VRing.e");
//RULE No.1.1.1.a & b
          fstream.open (filename.c_str(), ofstream::out);
//open file
//RULE No.1.1.1.c

          //TAG_Rule_VerificationRing CVRing(CInterface);
          CVRing.init(CInterface);
          CVRing.CreatCode(fstream); //Call VerificationRing RULES
                                     // to add function to file
                                //RULE No.1.1.1.d
          break;

     case InputPackage:
```

```
          filename = (CInterface->Get_ProjectName() += "_IPackage.e");
            //open file
          fstream.open (filename.c_str(), ofstream::out);
     CIPackage.init(CInterface);
          CIPackage.CreatCode(fstream);
          break;

     case Scoreboard:
          filename = (CInterface->Get_ProjectName() += "_Scoreboard.e");
          fstream.open (filename.c_str(), ofstream::out);
     CSCBoard.init(CInterface);
          CSCBoard.CreatCode(fstream);
          break;

     case Coverage:
          filename = (CInterface->Get_ProjectName() += "_Coverage.e");
            fstream.open (filename.c_str(), ofstream::out);
            CCoverage.init(CInterface);
          CCoverage.CreatCode(fstream);
          break;

     case Checker:
          filename = (CInterface->Get_ProjectName() += "_Checker.e");
          fstream.open (filename.c_str(), ofstream::out);
     CChecker.init(CInterface);
          CChecker.CreatCode(fstream);
          break;

     case GlueLogic:
          filename = (CInterface->Get_ProjectName() += "_GlueLogic.e");
          fstream.open (filename.c_str(), ofstream::out);
     CGLogic.init(CInterface);
          CGLogic.CreatCode(fstream); //Call GlueLogic RULES to add
                                      //function to file
          break;

     case InjectPort:
          filename = (CInterface->Get_ProjectName() +=
"_InjectPort.e");//RULE No.1.3.1 a & b
            fstream.open (filename.c_str(), ofstream::out);      //RULE
            No.1.3.1 c
          CInjePort.init(CInterface);
            CInjePort.CreatCode(fstream);
            //RULE No.1.3.1 d
          break;

     case Monitor:
          filename = (CInterface->Get_ProjectName() += "_Monitor.e");
            //RULE No.1.4.1 a & b
          fstream.open (filename.c_str(), ofstream::out);  //open file
//RULE No.1.4.1 c
          CMonitor.init(CInterface);
          CMonitor.CreatCode(fstream);
//RULE No.1.4.1 d
```

```
          break;

     case Other:

          list<string> MNameList;
                //  char* MName;
          string  FName;
          //MNum = CInterface.IMPackagePlus_ModuleNum();

                    //create modules
          MNameList = CInterface->Get_IMPackagePlus_ModuleName();

                list<string>::iterator i;
                for (i = MNameList.begin(); i != MNameList.end(); i++)
{
               //No.1.5.a
               tmp += "_";
               tmp += (*i).c_str();
               FName = (CInterface->Get_ProjectName() += tmp);
               filename = FName;
               filename += (".e");     //Rule No. 1.5.b
               //filename = (CInterface.GetProjectName() += tmp +=
".e");
               fstream.open (filename.c_str(), ofstream::out);  //open
file  //Rule No.1.5.c
               CEmulation.init(CInterface);
               CEmulation.CreatCode(fstream,FName);     //Rule No. 1.5.d
                }

          return 0;
          break;

//   default:
//        printf("Module Name Error\n");
//        return 1;


     }


     return 0;
}



int TAG_Rule_Partition::IMPackage()  //Rule No. 1.2.1
{

     if(CInterface->Get_IMPackage()) {  //use Inject-Monitor package
     //need generate Inject and Monitor module for testbench
          Create_InjectPort();    //call rule No. 1.3
          Create_Monitor();       //call rule No. 1.4

          if(CInterface->Get_IMPackagePlus()) { //I-M package plus
```

```
                if(GenerateFile(Other)) // Call Rule No. 1.5
                    return 0;
                else
                    return 1;

            }
        }
        else {

        }


        return 0;
}
```

# APPENDIX C: EXAMPLE OF OUTPUT TESTBENCH FILE

## F1. File atm_switch_vring.e

```
-------------------------------------------------------------------------
--------
-- Title          : Module Verification Ring
-- Project        : Testbench modules generation
-------------------------------------------------------------------------
--------
-- File           : atm_switch_vring.e
-- Author         : Tool ATG

-------------------------------------------------------------------------
--------
-- Description :
-------------------------------------------------------------------------
--------
    module: atm_switch_vring
    aspect: base

-------------------------------------------------------------------------
--------
-- Nom            : sys
-- Type           : struct
-- Particularite  : Extension
-- Description    : Permet d'attacher le testbench a l'environnement de
base de
--                  de Specman Elite.
-------------------------------------------------------------------------
--------
<'
extend sys {
event base_clk;
event driving_clk;     -- Horloge au front montant
event driving_clk_n;   -- Horloge au front descendant

event reset;           -- Evenememenet RESET
event unreset;         -- Evenement fin de reset

on reset {             -- Sur l'evenement Reset la fonction doReset a
                       -- appele
  doReset();
};
on unreset {           -- Sur l'evenement unReset la fonction doUnReset
                       -- a appele
  doUnReset();
};
doReset() is empty;
doUnReset() is empty;
};
'>
```

```
<'
unit atm_switch_VRing {
   config      : vring_configuration;

   keep config.parent == me;

   initTest() @sys.driving_clk is {
      wait [4] * cycle;
      emit sys.reset;
      wait [4] * cycle;
      emit sys.unreset;
      wait [4] * cycle;
      start vringReset();
   };

   killTest() @sys.driving_clk is {
      wait [config.time_bomb] * cycle;
      stop_run();
   };

   vringReset() @sys.driving_clk is empty;

   run() is also {
      emit sys.base_clk;
      start killTest();
      start initTest();
   };
};
'>


<'
struct vring_configuration{
   parent :atm_switch_VRing;
   period: uint;
   time_bomb :uint;
   keep soft period    == 5;
   keep soft time_bomb  == 1000000;

};
'>
```

## F2. File atm_switch_gluelogic.e

```
--------------------------------------------------------------------------------
--------
-- Title          : Module Verification Ring
-- Project        : Testbench modules generation
```

```
-------------------------------------------------------------------------
--------
-- File          : atm_switch_gluelogic.e
-- Author        : Tool ATG

-------------------------------------------------------------------------
--------
-- Description :
-------------------------------------------------------------------------
--------
    module: atm_switch_gluelogic
    aspect: gluelogic

-------------------------------------------------------------------------
--------
-- Nom           : sys
-- Type          : struct
-- Particularite : Extension
-- Description   : Permet d'attacher le testbench a l'environnement de
base de
--                 de Specman Elite.
-------------------------------------------------------------------------
--------

<'
    import atm_switch_VRing;
    import atm_switch_IPackage;
    import atm_switch_InjectPort;
    import atm_switch_Monitor;
    import atm_switch_generator_mgmt;
    import atm_switch_Scoreboard;
    --import atm_switch_Coverage;
    --import atm_switch_Checker;

'>



<'
extend sys {
   vring : atm_switch_VRing is instance;
   event base_clk is only {
      @base_clk; delay(vring.config.period)
   };

   doReset() is also {
      for each (gen) in vring.injectport do {
         gen.do_reset();
      };
      for each (mon) in vring.monitor do {
         mon.do_reset();
      };
      --vring.generator_mgmt.do_reset();
   };
```

```
   doUnReset() is also {

   };
};

'>


<'
extend atm_switch_VRing {
   injectport[4]: list of atm_switch_InjectPort is instance;
   monitor[4]: list of atm_switch_Monitor is instance;
   generator_mgmt: atm_switch_generator_mgmt is instance;
   sb: atm_switch_Scoreboard is instance;

   keep for each (g) in injectport  do {
      g.id     == index;
      g.parent == me;
   };
   keep for each (m) in monitor  do {
      m.id     == index;
      m.parent == me;
   };



   -- Fonction test : sert a injecter les cas de test
   test_squat()@sys.driving_clk is  empty;
   -- extension de la fonction initTest afin de demarrer le test
   initTest()@sys.driving_clk is also{
      start test_squat();
   };

};
'>

<'
extend atm_switch_InjectPort{
   parent: atm_switch_VRing;

   send_transaction(t: InjectPackage)@sys.driving_clk is also{
      parent.sb.add_transaction(t);
   };
};

'>



<'
extend atm_switch_Monitor{
   parent: atm_switch_VRing;
```

```
   get_transaction(t: MatchPackage)@sys.driving_clk is also{
      parent.sb.match(t);
   };
};

'>
```